João Manuel Gomes Moura

Master of Science in Computational Logic

Modular Logic Programming:

Full Compositionality and Conflict Handling for Practical Reasoning

Dissertação para obtenção do Grau de Doutor em

Engenharia Informática

Orientador: Carlos Augusto Isaac Piló Viegas Damásio,
Associate Professor, Universidade Nova de Lisboa

Júri

Presidente:	Professor José Júlio Alferes
Arguentes:	Professor José Pedro Cabalar Fernandez
	Docent Tomi Janhunen
Vogais:	Professor Carlos Augusto Isaac Piló Viegas Damásio
	Professor Ricardo Rocha

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

22 de Dezembro de 2016

Acknowledgements

First of all, I would like to express my deepest gratitude and appreciation to my supervisor, Professor Carlos Viegas Damásio, who has worked closely with me all these years. Being knowledgeable in so many research areas, I cannot recall how much I have learnt from him through an enormous number of discussions (not only during this PhD, but also during my Master studies and even before that, during my undergraduate studies). His genuine enthusiasm for new research ideas shows me an example of an extraordinarily dedicated scientist, which, I believe, will strongly influence my research approach and attitude for the rest of my career. I truly enjoy and appreciate the many long discussions that he always generously accommodated for when necessary. Furthermore, I am profoundly indebted to his constant support in all matters, not only in my studies, but also in the more bureaucratic ones.

Besides my supervisor, I am most grateful to Professor Thomas Eiter, who not only provided me with significant guidance for an important part of my PhD research, but also showed essential support and warm friendship. I truly appreciate all the precious scientific connections Thomas has helped to establish, which are undoubtedly crucial for my career, and our several interesting discussions and collaborations during these years.

I am very grateful to the members of my Technical Advisory Committee, Professors José Pedro Cabalar, Ana Moreira and José Júlio Alferes, who have contributed with essential suggestions on how to converge my writings into an appropriate PhD thesis. To Professor Luís Moniz Pereira goes also a warm thank you for all the interesting talks and for encouraging me to start my research career in the first place.

I thank Professors José Alferes and João Leite, for providing me with essential scientific background. I also thank Professors Luís Caires, Nuno Correia and Pedro Barahona who were always of great help with respect to the administration issues.

I acknowledge the Department of Informatics, Faculty of Sciences and Technologies, New University of Lisbon and the Center for Artificial Intelligence (CENTRIA) and NOVA-LINCS for providing me with a place and tools to work. I thank Sandra Raínha, Filipa Reis, Susana Pereira and Anabela Duarte for helping me with the several bureaucratic issues and with arranging my travels during these years.

I thank all my wonderful friends at FCT/UNL and specially Martin Slota, Han The Anh and Ana Sofia Gomes who have always been very supportive and good friends.

Finally, none of this would have been possible without the love and support of

my family. Starting with Cristina, who has always been by my side, loving and understanding me, and constantly supporting me through this challenging period. She forced me to work hard and keep up the pace whenever I showed signs of slacking but also rewarded me when she felt I was on track – a good manager. I am most indebted to my parents José Domingos and Arminda, who, throughout my life, have always been of the greatest and unconditional support in all matters; and to my brother Rui and my sister in law Maria who have always encouraged me to pursue higher studies and a career in science. To my nephews João and Inês who brought me great joy and to my grandmother Maria do Carmo, who always supported me, and managed to keep me entertained with her fun and youthful spirit, and without whom this journey would have been much harder.

This thesis is dedicated to all of them.

The work of João Moura was supported by grant SFRH/BD/69006/2010 from Fundação para a Ciência e Tecnologia (FCT) from the Portuguese Ministério do Ensino e da Ciência.

Abstract

With the recent development of a new ubiquitous nature of data and the profusity of available knowledge, there is nowadays the need to reason from multiple sources of often incomplete and uncertain knowledge. Our goal was to provide a way to combine declarative knowledge bases – represented as logic programming modules under the answer set semantics – as well as the individual results one already inferred from them, without having to recalculate the results for their composition and without having to explicitly know the original logic programming encodings that produced such results. This posed us many challenges such as how to deal with fundamental problems of modular frameworks for logic programming, namely how to define a general compositional semantics that allows us to compose unrestricted modules.

Building upon existing logic programming approaches, we devised a framework capable of composing generic logic programming modules while preserving the crucial property of compositionality, which informally means that the combination of models of individual modules are the models of the union of modules. We are also still able to reason in the presence of knowledge containing incoherencies, which is informally characterised by a logic program that does not have an answer set due to cyclic dependencies of an atom from its default negation. In this thesis we also discuss how the same approach can be extended to deal with probabilistic knowledge in a modular and compositional way.

We depart from the *Modular Logic Programming* approach in Oikarinen & Janhunen (2008); Janhunen *et al.* (2009) which achieved a restricted form of compositionality of answer set programming modules. We aim at generalising this framework of modular logic programming and start by lifting restrictive conditions that were originally imposed, and use alternative ways of combining these (so called by us) *Generalised Modular Logic Programs*. We then deal with conflicts arising in generalised modular logic programming and provide modular justifications and debugging for the generalised modular logic programming setting, where justification models answer the question: *Why is a given interpretation indeed an Answer Set?* and Debugging models answer the question: *Why is a given interpretation not an Answer Set?*

In summary, our research deals with the problematic of formally devising a generic modular logic programming framework, providing: operators for combining arbitrary modular logic programs together with a compositional semantics; We characterise conflicts that occur when composing access control policies, which are generalisable to our context of generalised modular logic programming, and ways of dealing with them syntactically: provided a unification for justification and debugging of logic programs; and semantically: provide a new semantics capable of dealing with incoherences. We also provide an extension of modular logic programming to a probabilistic setting. These goals are already covered with published work.

A prototypical tool implementing the unification of justifications and debugging is available for download from http://cptkirk.sourceforge.net.

Keywords: Logic Programming, Answer Set Programming, Paracoherence, Paraconsistency, Modular Logic Programming, Probabilistic Logic Programming, Bayesian Networks, Access Control Policies.

Sumário

A natureza ubíqua que os dados têm hoje em dia, bem como a quantidade de conhecimento disponível, obrigam-nos a desenvolver formas de raciocinar a partir de múltiplas fontes de conhecimento muitas vezes incompletas e incertas. Utilizamos bases de conhecimento declarativas, representadas na forma de programas em lógica modulares utilizando a semântica de programação por conjuntos de respostas.

O nosso objetivo nesta tese é fornecer uma maneira de combinar estas bases de conhecimento assim como as computações individuais previamente derivadas delas na forma de modelos lógicos, sem que seja necessário recalcular esses resultados para proceder à composição dos módulos e sem ter que saber explicitamente as regras dos módulos originais que produziram tais resultados. Isso colocou-nos muitos desafios, tais como perceber qual a forma de lidar com os problemas fundamentais das abordagem de Programação em Lógica Modular existentes na literatura, ou seja, como definir uma semântica composicional geral que nos permita compor módulos sem restrições.

Partindo de abordagens existentes no contexto de LP, desenvolvemos uma framework nova capaz de lidar com composição de módulos genéricos em LP, preservando a propriedade crucial de composicionalidade que, informalmente, significa que a combinação dos modelos dos módulos individuais são os modelos da união dos módulos. Queremos ainda ser capazes de raciocinar na presença de conhecimento contendo incoerências: Dizemos que um programa em lógica que não tem um conjunto de resposta devido a dependências cíclicas que um átomo tem da sua negação por omissão (default negation) é incoerente. Nesta tese discutimos também a forma como esta mesma abordagem pode ser estendida para lidar com conhecimento probabilístico.

Partimos da abordagem de Programação em Lógica Modular (MLP) por Oikarinen & Janhunen (2008); Janhunen *et al.* (2009) que disponibiliza uma forma restrita de composicionalidade para estes módulos. Outro objetivo nosso foi o de generalizar esta abordagem e começámos por resolver os problemas que lhe detectámos. Lidamos de seguida com os conflitos que surgem em GMLP e fornecemos modelos para justificações e modelos para debugging para GMLP. Modelos de justificação são respostas à pergunta: *Por que é que uma determinada interpretação é de facto um conjunto de resposta?* E modelos de debugging são respostas à pergunta: *Por que é que uma determinada interpretação não é um conjunto de resposta?*

Em resumo, a nossa investigação aborda a problemática da elaboração formal de uma framework genérica de programação em lógica modular. Fornecemos: operadores para combinar programas em lógica modular arbitrários juntamente com uma semântica composicional; Caracterizamos os conflitos que ocorrem durante a composição de políticas de controlo de acesso, que são generalizáveis a GMLP e maneiras de lidar com estes conflitos de forma sintáctica: contribuímos com uma

unificação de duas abordagens existentes na literatura para justificação e debugging de programas em lógica; e de forma semântica: contribuímos com uma nova semântica capaz de lidar com incoerências. Também fornecemos uma extensão da abordagem de programação em lógica modular para uma contexto probabilistico. Estas metas foram todas atingidas e correspondem a trabalho publicado. Um protótipo que implementa a unificação das abordagens de justificações e debugging para programas em lógica está disponível para download em http://cptkirk.sourceforge.net.

Palavras Chave: Programação em Lógica, Programação por conjuntos de resposta, Paracoerência, Paraconsistencia, Modularidade de Programas em Lógica, Redes Bayesianas, Políticas de Controlo de Acesso.

Contents

List of Figures and Tables

Part I

Introduction

Chapter 1

Introduction

1.1 Introduction

With the recent development of a new ubiquitous nature of data and the profusity of available knowledge, there is nowadays the need to reason from multiple sources of often incomplete and uncertain knowledge. Our goal was to provide a way to combine declarative knowledge bases – represented as logic programming (LP) modules under the answer set semantics – as well as the individual results one already inferred from them, without having to recalculate the results for their composition and without having to explicitly know the original logic programming encodings that produced such results. The semantics of logic programming is given by sets of models known as Answer Sets. This posed us many challenges such as how to deal with fundamental problems of Modular Logic Programming (MLP), namely how to define a fully generic compositional semantics that would allows us to compose unrestricted modules.

Building upon existing logic programming approaches, we devised a novel framework capable of composing generic logic programming modules while preserving the crucial property of compositionality.

Compositionality (informally): The combination of models of individual modules are the models of the union of modules.

However, due to non-monotonicity, programs may be incoherent i.e., lack a model. Nonetheless, there are many cases when this is not intended and one might want to draw conclusions also from an incoherent program, e.g., for debugging purposes, or in order to keep a system (partially) responsive in exceptional situations; in particular, if the contradiction or instability is not affecting the parts of a system that intuitively matter for a reasoning problem.

Incoherence (informally): A logic program that does not have an answer set due to cyclic dependencies of an atom from its default negation is said to be incoherent.

In this thesis we also discuss how this modular approach can be extended to deal with probabilistic knowledge in a modular and compositional way.

We depart from the Modular Logic Programming (MLP) approach in Oikarinen & Janhunen (2008); Janhunen *et al.* (2009) which achieved a restricted form of compositionality of answer set programming modules. We aim at generalising MLP and start by lifting restrictive conditions that were originally imposed, and use alternative ways of combining these Generalised Modular Logic Programs (GMLP). We then deal with conflicts arising in GMLP and provide modular justifications and debugging for the GMLP setting,

Justification models (informally): Answer the question: *Why is a given interpretation indeed an Answer Set?*

Debugging models (informally): Answer the question: *Why is a given interpretation not an Answer Set?*

The **research questions** we posed basically motivated the two parts of our research and were, namely, for the first part and second parts:

"Do Modular Logic Programming frameworks have the sufficient expressiveness to serve as a basis to develop a fully compositional and modular knowledge representation framework? Are more general knowledge representation frameworks well suited to achieving full compositionality?"

"What are the general ways in which one can deal with the conflict that occur when combining knowledge from multiple sources?"

Because of these questions, the aim of this research was twofold.

1. We aimed at devising a general modular knowledge representation framework, attaining a full form of compositionality that was missing in the current state-of-the-art although several modular knowledge representation and reasoning frameworks exist, some of them having partial compositionality properties.

2. Given a full compositional knowledge representation framework – in the form of generalised modular logic programs – we aimed at studying and dealing with the conflicts that occur when composing knowledge originating from multiple knowledge bases.

There are, in general, two ways to identify and deal with said problems: **Syntactically**, by detecting the syntactical causes for why some *unintended* (respectively, *intended*) interpretation *is* (respectively, *is not*) a model; **Semantically**, by using semantics to avoid certain categories of conflicts. Such semantics have specific properties such that allow them to deal with inconsistencies and incoherencies, giving models to programs that suffer from these problems.

Shortcomings have been identified in the literature of Modular Logic Programming, namely the impossibility to compose modules with common output atoms and modules that are mutually dependent through positive cycles. When these restrictions are lifted, inconsistencies and conflicts arise when composing modules and, instead of accepting the explosive approach in the presence of contradiction, one might be interested in having different degrees of certainty associated to knowledge stored within each module and also in ways to attain paracoherency [1] with respect to knowledge. We believe that in the future, the broad acceptance of logic programming as a whole will greatly benefit from efforts on the integration of developments in these directions.

As a field of application for this framework, we bring these notions to access control policies, where such policies can be easily encoded in modular logic programming. We need to characterise the conflicts that arise when combining these policies, which can in turn be ported to the generalised setting of modular answer set programming, identifying and characterising its basic conflict types which can be studied according to several dimensions. Other notions can also be ported to this application field: e.g., giving a probability or degree of certainty to knowledge stored within a module (policy).

1.2 Motivation

The theoretical framework we developed potentially allows us to efficiently represent and process logic programs not as monolithic entities but rather as sets of modules. In this thesis, we resort to a compositional semantics allowing models from individual modules to be retained and composed. This is critical in any real world scenario, as the amount of information is too big for one to afford having to, whenever adding new information, recalculate results of modules that potentially have already been calculated. Furthermore there are some scenarios where the logic programming encoding's owner might not wish to divulge its contents but only part of its results. We are also able to provide justifications for why a wanted interpretation is not an answer set (AS) and for why an interpretation that is not desired to be a model, is indeed an answer set.

Most real world databases contain data which is not certain and in order to be able to reason from such data, there is the need to capture this knowledge by means of systems that are formally well defined. This can be achieved by using e.g., probabilistic knowledge bases.

Again, in a real world application, one often faces the need to validate our logic programs as well as to identify and then deal with conflicts. The Logic Programming community is nowadays very well aware of this need Eiter *et al.* (2010a); Gebser *et al.* (2008); Shchekotykhin (2014); Pemmasani *et al.* (2004); Pontelli *et al.* (2009)

[1] We use the term *paracoherent* reasoning to distinguish between paraconsistent reasoning and reasoning from incoherent programs.

and a way of theoretically characterising such problems and conflicts has also been one of our focus.

Why Answer Set Programming? Answer Set Programming (ASP) is a prime formalism for non-monotonic knowledge representation and reasoning, mainly because of the existence of efficient solvers and well established relationships to common non-monotonic logics. It is a declarative logic programming paradigm with a model theoretic semantics, where problems are encoded into a logic program using rules, and its models, called answer sets (or stable models) due to Gelfond & Lifschitz (1988), encode solutions. Over the last two decades, the paradigm of answer set programming has been receiving ever increasing attention from the logic programming communities Eiter *et al.* (2001); Baral (2003); Lifschitz (2002); Marek & Truszczynski (1999); Niemelä (1998). A good overview is provided in Brewka *et al.* (2011);

Real World Scenario: The Problem of Access Control Policies The emergence of technologies such as service-oriented architectures and cloud computing has allowed us to perform information services more efficiently and effectively. Access control is an important mechanism for achieving security requirements in such information systems, which are described by means of access control policies (ACPs).

However, these security requirements cannot be guaranteed when conflicts occur in these ACPs. Furthermore, the design and management of imperative access control policies is often error-prone due not only to the lack of a logical and formal foundation but also to the lack of automated conflict detection and resolution.

Identifying their basic conflict types and characterising them formally, would allow the automatic identification of such conflicts among other reasoning tasks. This need for characterising conflicts comes not only from the non-monotonic nature of access control but also from the fact that some policies are distributed and as such they can derive conflicting conclusions. Answer set programming, together with these characterisations, has the potential to vastly improve the ease of designing and maintaining complex systems of access control policies.

Motivating Use-Case: Hospital Access Control Policy We take and adapt a well-known use case of a set of modular access control policies for a hospital Bonatti *et al.* (2002). We split the original example into three sub-examples for ease of presentation:

Example 1.2.1 *Consider a hospital composed of three departments, namely Radiology, Surgery, and Medicine. Each of the departments is responsible for granting access to data under their (possibly overlapping) authority domains. The statements made by the departments are unioned, meaning the hospital considers an access as authorised if any of the department policies states so.*

For privacy regulations, however, the hospital will not allow any access (even if authorised by the departments) to lab tests data unless there is patient consent for

that, stated by policy $P_{consents}$. However, the patient should only be asked for consent to divulge documents which can indeed be allowed for access by the hospital.

Accordingly, lab tests data will be released only if both the hospital authorizes the release and the interested patient consents to it.

The departments are also responsible for discharging their patients. △

As an example of policy modules, we zoom in on $P_{consents}$ and $P_{medicine}$.

Example 1.2.2 *$P_{consents}$ reports accesses to laboratory tests for which there is patient consent. Authorisations in $P_{consents}$ are collected by the hospital administration by means of forms that patients fill in when admitted. Patients' consents can refer to single individuals (e.g., John Doe can individually point out that his daughter Jane Doe can access his tests) as well as to subject classes (e.g., research labs and hospitals), and can refer to single documents or to classes of them.*

Authorisations specified for subject/object classes are propagated to individual users and documents by classical hierarchy-based derivation rules. △

Example 1.2.3 *Policy $P_{medicine}$ of the medical department is composed of the policies of its two divisions, Cardiology and Oncology, and of a policy $P_{administration}$ specified by the central administration of the department. The Oncology division can revoke authorisations in $P_{administration}$, regarding data related to clinic trials, by issuing negative-administration-allows predicates in $P_{oncology}$.*

In addition, each of the divisions can specify further authorisations (policies P_{onc} and $P_{cardiology}$), whose scope is restricted to objects in their respective domains. △

Note that some policies refer to the same subjects, e.g., *discharged* in $P_{oncology}$ and $P_{cardiology}$, and that 1.2.1 has mutual dependencies in the conditions for allow and consents. Both issues are problematic with current modular logic programming technologies.

1.3 Thesis Results Summary

In short, the output of this PhD thesis is a logic framework generalising modular logic programming results and providing: operators for combining arbitrary LP modules, compositional semantics for GMLP, characterisation of conflicts that occur when composing modules and ways of dealing with them semantical (paracoherent semantics) and syntactically by unifying justifications and debugging LP modules.

- As a possible real world application, we aimed at identifying and formalising possible causes of conflict in access control policies, as well as presenting reasonable ways of detecting and fixing them either by:
  - Using paracoherent semantics (e.g., our $\mathcal{SEQ}$-model semantics Eiter *et al.* (2010b); Amendola *et al.* (2015)), or,

  - By providing means for applying syntactic changes to faulty programs by finding justifications and debugging models Damásio *et al.* (2015).

- Using generalised modular logic programming (GMLP) Damásio & Moura (2014, 2015) for this purpose yields different types of basic conflicts in access control programs.

  - We characterised them in terms of the notions of strong equivalence of logic programs, as well as that of default logic, in Moura (2012).

- These characterisations enable the detection of conflicts and allow this to be done automatically. They are, overall, flexible enough to be extended to detecting which types of conflicts are generated, as well as to trace them back to the source, potentially identifying leaks in ACPs.

  - A prototypical tool implementing the unification of justifications and debugging is available for download at http://cptkirk.sourceforge.net.

- Furthermore, one can use probability distributions for detecting errors or conflicts that are more probable. Causal probability tables must be calculated for this purpose and then encode these into our modular P-Log framework Damásio & Moura (2011).

Chapter 2

Preliminaries: Logic programs and Modular Frameworks

In this Chapter 2, we introduce relevant formal background, namely, in the following two sections where we start by presenting in Section 2.1.1 an overview of relevant work and introduce necessary fundamental concepts and formalisms in the context of logic programming and answer set programming, and then in Section 2.2 we introduce their modular aspects.

2.1 Logic Programming Formalisms

We consider programs in a function-free first-order language (including at least one constant). As for terms, strings starting with uppercase (respectively, lowercase) letters denote variables (respectively, constants). An atom is an expression of the form $p(t_1,\ldots,t_n)$, where p is a predicate symbol and each t_i is a term. A literal is either an atom a or an expression of the form $-a$, where $-$ denotes strong negation (also interchangeably denoted by $\neg$).

2.1.1 Answer set programming paradigm

Normal Logic Programs in the answer set programming paradigm are formed by finite sets of rules r having the following syntax:

$$L_1 \leftarrow L_2,\ldots,L_m,not\ L_{m+1},\ldots,not\ L_n.\ \ (n \geq m \geq 0) \tag{2.1}$$

Where each L_i is a literal. A **disjunctive rule** r is of the form:

$$L_1 \vee \ldots \vee L_l \leftarrow L_{l+1},\ldots,L_m,not\ L_{m+1},\ldots,not\ L_n.\ \ (n \geq m \geq l \geq 0) \tag{2.2}$$

where each L_i is a literal[1] without the occurrence of function symbols – arguments are either variables or constants of the logical alphabet.

[1] A statement letter or a negation of a statement letter Mendelson (1987).

Literals (respectively, rules, programs) are **ground** if they are variable-free. Non-ground literals (respectively, rules, programs) amount to their ground instantiations, i.e., all instances obtained by substituting variables with constants from the (implicit) language. The ground instantiation of a program P is denoted by $Gr(P)$. By $At(P)$ we denote the set of all (ground) atoms occurring in $Gr(P)$.

Definition 2.1.1 (Choice Rules) *The syntax of logic programs has been extended with other constructs, namely weighted and* **choice rules** *Niemelä (1998). In particular, choice rules have the following form:*

$$\{A_1,\ldots,A_n\} \leftarrow B_1,\ldots B_k,\ not\ C_1,\ldots,\ not\ C_m.\ (n \geq 0) \tag{2.3}$$

where $A_1,\ldots,A_n,\ B_1,\ldots B_k,\ C_1,\ldots,\ C_m$ *are atoms.* ▲

As observed by Oikarinen & Janhunen (2008), the atoms in the consequents of choice rules possessing multiple atoms can be freely split without affecting their semantics. When splitting such rules into n different rules

$$\{A_i\} \leftarrow B_1,\ldots B_k, not\ C_1,\ldots, not\ C_m.\ \text{where}\ 1 \leq i \leq n$$

the only concern is the creation of n copies of the implicants $B_1,\ldots B_k, not\ C_1,\ldots, not\ C_m$. However, new atoms can be introduced to circumvent this. There is a translation of these choice rules to normal logic programs Ferraris & Lifschitz (2005), which we assume is performed throughout this thesis but that is omitted for readability. We, furthermore, deal only with ground programs and use variables as syntactic placeholders.

Intuitively, a Normal Logic Program is a logic program where negation is allowed in the bodies of rules and no disjunction is allowed (in the heads of rules).

Disjunctive Logic Programs (DLPs) are finite sets of disjunctive rules (over language Σ). A program P is called *normal* (respectively, *positive*) if each $r \in P$ is normal (respectively, positive); P is *constraint-free*, if P contains no constraints.

Example 2.1.1 (Disjunctive Logic Program) *Consider the following disjunctive logic program* $P = \{assistant \vee student \leftarrow not\ professor.\ discount \leftarrow student, not\ assistant.\}$. *It intuitively captures that some department members who are not known to be professors are assistants or students, and a student who is not known to be assistant gets a discount for coffee.*

△

Rule Notation Considering a rule r of the forms (2.1) or (2.2), let:

$$Head(r) = L_1$$

be the literal in the head of a non-disjunctive rule and:

$$Head(r)^D = \{L_1,\ldots,L_l\},$$

be the set with all literals in the head of a disjunctive rule. Let:

$$Body^+(r) = \{L_2, \ldots, L_m\}$$

$$(\text{or } Body^+(r)^D = \{L_{l+1}, \ldots, L_m\} \text{ for the disjunctive case})$$

be the set with all positive literals in the body, let:

$$Body^-(r) = \{L_{m+1}, \ldots, L_n\}$$

be the set containing all negative literals in the body, and let:

$$Body(r) = \{L_2, \ldots, L_n\}$$

be the set containing all literals in the body.

If a program is positive we will omit the superscript in $Body^+(r)$. When the context is clear we will simply use $Head(r)$ as a set of atoms for disjunctive rules or as a single atom for non-disjunctive rules and the same for the corresponding literal indexes in the bodies of non-disjunctive and disjunctive rules.

Next we define different rule types, namely facts, constraints as well as normal and positive rules, in terms of their heads and bodies:

Definition 2.1.2 (Rule Types) *A rule r of forms (2.1) or (2.2) is called:*

(i) *a* fact, *if* $Body(r) = \emptyset$ *(in which case the symbol* $\leftarrow$ *is usually omitted),*

(ii) *an* integrity constraint *(IC), if* $Head(r) = \emptyset$*,*

(iii) normal, *if* $|Head(r)| \leq 1$, *and*

(iv) positive, *if* $Body^-(r) = \emptyset$*.*

▲

Let us denote an **interpretation** I by the set of atoms satisfied by I. Note here that given a program P and an interpretation I, if an integrity constraint $c \in Gr(P)$ is applicable with respect to I, the fact that $Head(c) = \emptyset$ implies $I \not\models c$, and thus I cannot be an answer set of P.

Definition 2.1.3 (Herbrand Base, Interpretations and Least Models) *Given a set of literals J let its default negation be not* $J = \{a \mid not\ a \in J\} \cup \{not\ a \mid a \in J \wedge \forall_b\ a \neq not\ b\}$*. The* **Herbrand Base** $At(P)$ *(frequently denoted as* $\mathbb{H}_P$*) of a program P is formed by the set of atoms occurring in it.*

A **two-valued interpretation** *I corresponds to the partial interpretation* $I \cup not\ (\mathbb{H}_P \setminus I)$.

The **least model** $LM(P)$ *of a definite program P is the least fixed point of operator*

$$T_P(I) = \{Head(r) \mid r \in P \wedge Body(r) \subseteq I\},$$

where I is a two-valued interpretation which is a subset of $\mathbb{H}_P$ *specifying the true atoms, and a partial interpretation is a subset of* $\mathbb{H}_P \cup not\ \mathbb{H}_P$ *(absent literals are undefined).* ▲

Definition 2.1.4 (Applicable and Blocked Rules) *Given a program P and an interpretation I for P, whenever $Body^+(r) \subseteq I$ and $Body^-(r) \cap I = \emptyset$, for a rule $r \in Gr(P)$, we say that r is* **applicable** *with respect to I, and* **blocked** *with respect to I otherwise.* ▲

Definition 2.1.5 (Support) *For a program P, an interpretation I for P, and $G \subseteq At(P)$, we call a rule $r \in Gr(P)$ a support for G with respect to I if r is applicable with respect to I, $Head(r) \cap G \neq \emptyset$, and $Head(r) \cap I \subseteq G$.*

We call G ***supported*** *by P with respect to I if there is some support $r \in Gr(P)$ for G with respect to I.*

Furthermore, G is ***unsupported*** *by P with respect to I if G is not supported by P with respect to I.* ▲

Next we define three semantics for logic programs which we will use thoroughly in this thesis, namely the well-founded model semantics, the answer set semantics and the partial stable model semantics.

Gelfond-Lifshitz Reduct: Answer Set Semantics The stable model semantics of is defined via the reduct operation Gelfond & Lifschitz (1988, 1990). Given an interpretation M (a set of ground atoms), the reduct P^M of a program P with respect to M is the program:

$$P^M = \{Head(r) \leftarrow Body^+(r) \mid r \in P, Body^-(r) \cap M = \emptyset\} \tag{2.4}$$

The interpretation M is a **stable model** of P iff $M = LM(P^M)$, where $LM(P^M)$ is the least model of reduct P^M. The new terminology of **answer set** appeared later in Lifschitz (1999) and it is now generally used for describing the variant based on 3-valued Herbrand Models (consistent sets of ground literals $M \subseteq \mathbb{H}_P \cup not\ \mathbb{H}_P$), with incomplete information of the world. The term stable models is nowadays usually reserved for programs without strong negation ("$-$"), and viewed as 2-valued models with complete information of the world.

Well-Founded Model Semantics (van Gelder *et al.* (1991)) The **Well-Founded Model** of P is $T \cup not\ F$ where T and F are interpretations such that $T \cap F = \emptyset$, T is the least fixed point $T = \Gamma(\Gamma(T)) = \Gamma^2(T)$ and $F = \mathbb{H}_P \setminus \Gamma(T)$.

The well-founded model semantics is the most prominent approximation of the answer set semantics but, unlike it, a well-founded model is defined for every normal logic program.

Partial Stable Models Note that every answer set is a fixed point of Γ^2 (but not the other way around) and that fixed points of Γ^2 are the partial stable models (PSMs) of Przymusinski (1990, 1991b). Compared to answer sets, partial stable models conservatively extend the class of programs for which an acceptable model exists; in particular, every non-disjunctive program has some partial stable model, while it may lack an answer set.

2.1.2 Equilibrium Logic

Intuitionistic logic is a system of symbolic logic that differs from classical logic inasmuch as it replaces the traditional concept of truth with the concept of constructive provability. For example, in classical logic, propositional formulae are always assigned a truth value from the two element set of trivial propositions $\{\top, \bot\}$ (*true* and *false* respectively) regardless of whether we have direct evidence for either case. This is referred to as the 'law of excluded middle', because it excludes the possibility of any truth value besides 'true' or 'false'. In contrast, propositional formulae in intuitionistic logic are not assigned any definite truth value at all and instead are only considered *true* when one has direct evidence for them, hence proof. Operations in intuitionistic logic therefore preserve justification, with respect to evidence and provability, rather than classical truth-valuation.

The definition of answer set in Section 2.1.1 uses the GL-reduct, and thus in a sense has an operational flavour. This raised the question whether a characterisation of answer sets in terms of a suitable logic is possible; and as constructibility of answer sets by rules is crucial, whether in particular (a variant of) intuitionistic logic, could serve this purpose. David Pearce showed that the answer is positive and presented *equilibrium logic* Pearce (2006a); Pearce & Valverde (2008), which is a natural nonmonotonic extension of Heyting's *logic of here-and-there* (HT Logic) Heyting (1930). The latter is an intermediate logic between (full) intuitionistic and classical logic, and it coincides with 3-valued Gödel logic. As it turned out, HT-logic serves as a valuable basis for characterising semantic properties of answer set semantics and equilibrium logic can be regarded as a logical reconstruction of answer set semantics that has many attractive features.

As such, HT-logic considers a full language $\mathcal{L}_{\pm}$ of formulas built over a propositional signature Σ with the connectives $\neg$, $\wedge$, $\vee$, $\rightarrow$, and $\bot$. We restrict our attention here to formulas of the form:

$$b_1 \wedge \ldots \wedge b_m \wedge \neg b_{m+1} \wedge \ldots \wedge \neg b_n \rightarrow a_1 \vee \cdots \vee a_l, \tag{2.5}$$

which correspond in a natural way to rules of form (2.2) where for $l = 0$, the formula $a_1 \vee \cdots \vee a_l$ is $\bot$; every program P corresponds then similarly to a theory (set of formulas) Γ_P.

Example 2.1.2 *For example, program P_1:*

$$P_1 = \{a \leftarrow b;\ b \leftarrow not\ c;\ c \leftarrow not\ a\}$$

corresponds to the theory:

$$\Gamma_{P_1} = \{b \rightarrow a;\ \neg c \rightarrow b;\ \neg a \rightarrow c\}$$

while program P_2 next:

$$P_2 = \{b \vee c \leftarrow not\ a;\ d \leftarrow c, not\ b\}$$

corresponds to the following theory:

$$\Gamma_{P_2} = \{\neg a \to b \vee c;\ \neg b \wedge c \to d\}$$

△

In the rest of this thesis, we tacitly use this correspondence.

As a restricted intuitionistic logic, HT can be semantically characterised by Kripke models, in particular using just two worlds, namely "*here*" and "*there*" (assuming that the *here* world is ordered before the *there* world).

An *HT-interpretation* is a pair (X,Y) of interpretations $X,Y \subseteq \Sigma$ such that $X \subseteq Y$; it is *total*, if $X = Y$. Intuitively, atoms in X (the *here* part) are considered to be true, atoms not in Y (the *there* part) to be false, while the remaining atoms (from $Y \setminus X$) are undefined.

Assuming that $X \models \phi$ denotes **satisfaction** of a formula ϕ by an interpretation X in classical logic, satisfaction of ϕ in HT-logic (thus, an HT-model), denoted $(X,Y) \models \phi$, is defined recursively as follows:

1. $(X,Y) \models a$ if $a \in X$, for any atom a,
2. $(X,Y) \not\models \bot$,
3. $(X,Y) \models \neg\phi$ if $Y \not\models \phi$ (that is, Y satisfies $\neg\phi$ classically),
4. $(X,Y) \models \phi \wedge \psi$ if $(X,Y) \models \phi$ and $(X,Y) \models \psi$,
5. $(X,Y) \models \phi \vee \psi$ if $(X,Y) \models \phi$ or $(X,Y) \models \psi$,
6. $(X,Y) \models \phi \to \psi$ if (i) $(X,Y) \not\models \phi$ or $(X,Y) \models \psi$, and (ii) $Y \models \phi \to \psi$.

Note that the condition in item 3 is equivalent to $(X,Y) \models \phi \to \bot$, thus we can view this intuitionistic form of negation $\neg\phi$ as implication $\phi \to \bot$. Then, an HT-interpretation (X,Y) is a **model** of a theory Γ, denoted $(X,Y) \models \Gamma$, if $(X,Y) \models \phi$ for every formula $\phi \in \Gamma$.

As regards negative literals and rules: given a HT-interpretation (X,Y), for an atom a it holds that $(X,Y) \models \neg a$ iff $a \notin Y$, and $(X,Y) \models r$ for a rule r of form (2.2) iff either $Head(r) \cap X \neq \emptyset$ or $Body^+(r) \not\subseteq Y$, or $Body^-(r) \cap Y \neq \emptyset$.

Definition 2.1.6 (Brave and Cautious Reasoning) *A formula φ is a brave consequence of a theory T, symbolically $T \mid\!\sim_b \varphi$, if and only if some equilibrium model of T satisfies φ. Dually, φ is a skeptical consequence of T , symbolically $T \mid\!\sim_s \varphi$, if and only if all equilibrium models of T satisfy φ. The basic reasoning tasks in the context of equilibrium logic are the following decision problems:*

- *Decide whether a given theory T possesses some equilibrium model.*
- *Given a theory T and a formula φ, decide whether $T \mid\!\sim_b \varphi$ holds.*

- *Given a theory T and a formula φ, decide whether $T \mid\sim_s \varphi$ holds.*

The first task is called the consistency problem; the second and third tasks are respectively called brave reasoning and skeptical reasoning. ▲

Brave reasoning – to decide whether there is an answer set containing a specific atom – is know to be in complexity class NP, whereas cautious reasoning – deciding whether a specific atom is in all the answer sets – is in co-NP. We recall that NP is the set of decision problems solvable in polynomial time by a theoretical nondeterministic Turing machine and, conversely, co-NP is the set of decision problems where the "no" instances can be accepted in polynomial time by a theoretical nondeterministic Turing machine.

SE-Model In terms of the GL-reduct, we have $(X,Y) \models P$ for a program P iff $Y \models P$ and $X \models P^Y$ Turner (2003a).

Definition 2.1.7 (Equilibrium Models) *A total HT-interpretation (Y,Y) is an **equilibrium model** ($\mathcal{EQ}$-model) of a theory Γ, if $(Y,Y) \models \Gamma$ and for every HT-interpretation (X,Y), such that $X \subset Y$, it holds that $(X,Y) \not\models \Gamma$; the set of all $\mathcal{EQ}$-models of Γ is denoted by $\mathcal{EQ}(\Gamma)$.* ▲

The equilibrium models of a program P are then those of Γ_P, i.e., $\mathcal{EQ}(P) = \mathcal{EQ}(\Gamma_P)$. For further details and background see, e.g., Pearce & Valverde (2008).

Example 2.1.3 *For program P:*

$$P = \{b \vee c \leftarrow not\ a;\ d \leftarrow c, not\ b\},$$

the sets[2] *$(\emptyset,a)$, (a,a), (b,b), $(\emptyset,ab)$, (a,ab), (b,bc), (c,bc), (cd,cd) are all HT-models (X,Y) of the corresponding theory Γ_P.*

The equilibrium models of P resp. Γ_P are (b,b) and (cd,cd), i.e., $\mathcal{EQ}(P) = \mathcal{EQ}(\Gamma_P) = \{(b,b),(cd,cd)\}$. △

In the previous example, the program P has the answer sets $I_1 = \{b\}$ and $I_2 = \{c,d\}$, which amount to the equilibrium models (b,b) and (cd,cd), respectively. In fact, the answer sets and equilibrium models of a program always coincide.

Proposition 1 (Pearce (2006a)) *For every program P and $M \subseteq At(P)$, it holds that $M \in \mathcal{AS}(P)$ iff (M,M) is an $\mathcal{EQ}$-model of Γ_P.* ∘

In particular, as $\mathcal{AS}(P) = MM(P)$ for any positive program P, we have $\mathcal{EQ}(P) = \{(M,M) \mid M \in MM(P)\}$ in this case.

As stated before, we call a logic program ***incoherent***, if it lacks answer sets due to cyclic dependency of atoms between each other by rules through negation; that is, no answer set (equivalently, no equilibrium model) exists even if all constraints are dismissed from the program.

[2] We write (as common) sets $\{a_1,a_2,\ldots,a_n\}$ as juxtaposition $a_1a_2\cdots a_n$ of their elements.

Example 2.1.4 *Consider the barber paradox, which can be regarded as an alternative form of Russell's famous paradox in naive set theory*[3]*: In some town, the barber is a man who shaves all men in town, and only those, who do not shave themselves. The paradox arises when we ask "Who shaves the barber?". Assuming that Joe is the barber, the knowledge about who is shaving him is captured by the logic program*

$$P = \{a \leftarrow not\ a\},$$

where a stands for shaves(joe, joe).

The HT-models of this program are $(\emptyset, a)$ *and* (a, a)*; the single total HT-model is* (a, a)*, which however is not an equilibrium model. Similarly, the program*

$$P = \{a \leftarrow b;\ b \leftarrow not\ a\}$$

has the HT-models $(\emptyset, a)$*,* $(\emptyset, ab)$*,* (a, a)*,* (a, ab)*, and* (ab, ab)*; likewise, the total HT-models* (a, a) *and* (ab, ab) *are not equilibrium models.* △

2.1.3 Paraconsistent and Paracoherent reasoning

As we have seen before, because of non-monotonicity, answer set programs may be incoherent, i.e., lack an answer set due to cyclic dependencies of an atom from its default negation. Nevertheless, there are many cases where this is not intended and one might want to draw conclusions also from an incoherent program, e.g., for debugging purposes, or in order to keep a system (partially) responsive in exceptional situations. This is akin to the principle of paraconsistency, where non-trivial consequences shall be derivable from an inconsistent theory. As so-called extended logic programs also may be inconsistent in the classical sense, i.e., they may have the inconsistent answer set as their unique answer set, we use the term *paracoherent reasoning* to distinguish between paraconsistent reasoning and reasoning from incoherent programs.

Both types of reasoning from answer set programs have been studied in the course of the development of the answer set semantics; for approaches on paraconsistent cf., e.g., Sakama & Inoue (1995a), Alcântara *et al.* (2005), Odintsov & Pearce (2005).

Numerous semantics for logic programs with non-monotonic negation can be considered as being paracoherent semantics. Ideally, such a semantics satisfies the following properties (desiderata):

(D1) Answer set coverage Every (consistent) answer set of a program corresponds to a model.

(D2) Congruence If a (consistent) answer set exists for a program, then all models correspond to an answer set.

(D3) Classical coherence If a program has a classical model, then it has a model.

[3] Namely, that the set of all sets that are not members of themselves can not exist.

Widely-known semantics, such as 3-valued stable models (Przymusinski (1991b)), L-stable models (Eiter *et al.* (1997a)), revised stable models (Pereira & Pinto (2005)), regular models (You & Yuan (1994)), and pstable models (Osorio *et al.* (2008)), satisfy only part of these requirements. Semi-stable models (Sakama & Inoue (1995a)) however, satisfy all three properties and thus are the prevailing paracoherent semantics.

Despite the model-theoretic nature of ASP, semi-stable models have been defined by means of a program transformation, called epistemic transformation and a semantic characterisation in the style of equilibrium models for answer sets Pearce & Valverde (2008) was still missing in the literature until recently. Such characterisation was desired because working with program transforms becomes cumbersome, if properties of semi-stable models should be assessed; and moreover, while the program transformation is declarative and the intuition behind it is clear, the interaction of the rules does not make it easy to understand or see how the semantics works in particular cases.

Example 2.1.5 (Semi-Stable Models) *Consider the following logic program P_1:*

$$P_1 = \{a \leftarrow not\ a\},$$

its epistemic transformation P_1^k (presented in depth in Chapter 7), has the single answer set $M = \{Ka\}$; hence, $\{Ka\}$ is the single semi-stable model of P, in which a is believed true. For program

$$P_2 = \{b \leftarrow not\ a\},$$

its epistemic transformation P_2^k has the answers sets $M_1 = \{Ka\}$ and $M_2 = \{\lambda_1, b\}$; as[4] $gap(M_1) = \{a\}$ and $gap(M_2) = \emptyset$, among them M_2 is maximal canonical[5], and hence $M_2 \cap \Sigma^K = \{b\}$ is the single semi-stable model of P_2. This is in fact also the unique answer set of P_2. Note that atoms marked with λ are auxiliary.

Finally, the epistemic transformation of

$$P_3 = \{b \vee c \leftarrow not\ a;\ d \leftarrow c, not\ b\}$$

has answer sets $M_1 = \{\lambda_{r_1,1}, b\}$, $M_2 = \{\lambda_{r_1,2}, c,\ \lambda_{r_2,1}, d\}$, $M_3 = \{\lambda_{r_1,2}, c, Kb\}$, and $M_4 = \{Ka\}$, as may be checked using a solver.

Among them, as $gap(M_1) = gap(M_2) = \emptyset$ while M_3 and M_4 have nonempty gap, M_1 and M_2 are maximal canonical and hence the semi-stable models of P_3; they correspond with the answer sets of P_3, $\{b\}$ and $\{c, d\}$, as expected. △

For a study of the semi-stable model semantics, we refer to Sakama & Inoue (1995b); notably,

[4] Considering *gap* to be the gap between believed and (derivably) true atoms.

[5] Meaning that there is no other model having a larger gap between believed and (derivably) true atoms, cardinality wise.

Proposition 2 (Sakama & Inoue (1995b)) *The SST -models semantics, given by SST(P) for arbitrary programs P, satisfies properties (D1)-(D3).* ○

Semi-stable models is the only semantics in the literature that satisfies these three properties.

2.2 Modular Frameworks: Overview and Formalisms

2.2.1 Multi Context Systems

A Multi Context System (MCS) Brewka & Eiter (2007b) is a collection of **contexts** that are linked using **bridge rules**. Each context has its own way of representing knowledge, including its its own syntax and semantics, i.e., a rather abstract and general notion of module. Bridge rules define how knowledge can be transferred between contexts. In MCSs, a model has the form of a collection of belief sets (called a belief state).

The semantics of MCSs that are of interest to us are as follows.

1. Equilibrium semantics (ES) defines intended models as exactly those belief states that, if viewed operationally, remain unchanged after first applying bridge rules and then applying contexts, hence the name of an equilibrium.

2. Minimal equilibrium semantics (MES) defines intended models as those equilibriums that are also minimal.

3. Grounded equilibrium semantics (GES) Brewka & Eiter (2007b) defines intended models as minimal equilibriums of a positive MCS obtained by reducing the original MCS. Reducing MCSs is similar (methodically and intent-wise) to the Gelfond/Lifschitz reduct Gelfond & Lifschitz (1988) used to define stable models.

The above semantics are motivated by everyday reasoning about a collection of contexts or agents. In MCSs, some knowledge is shared between different knowledge bases, while some knowledge is kept private/confidential. Note that justifications and, in particular, avoiding self-justifications was the main motivation behind the introduction of grounded equilibrium semantics for MCSs Brewka & Eiter (2007b). However, grounded equilibrium semantics (GES) is defined over MCSs in which all contexts are reducible. Thus, even one non-reducible context is enough to render GES non-applicable. Tasharrofi & Ternovska (2014) solved this by proposing an intermediate semantics capturing the robustness of ES, thus making it applicable to every MSC while, like GES does, dealing with the problem of self justified loops.

MCSs can not be said to be compositional in general though, as they take general contexts for which their combination might not be intuitive and need some operational mapping in the form of bridge rules. For instance, interpretations having underlying multi-valued logics might be hard to combine. We define this formally in Section 5.3.

2.2.2 Modular Logic Programming

The modular aspects of answer set programming have been clarified over the last years for normal logic programming Oikarinen & Janhunen (2008); Dao-Tran *et al.* (2009); Järvisalo *et al.* (2009); Damásio & Moura (2014); Babb & Lee (2012), for probabilistic logic programming Damásio & Moura (2011) and for disjunctive logic programs (DLP) Janhunen *et al.* (2009), describing how and when two program parts (modules) can be combined. In this section, we overview Oikarinen and Janhunen's logic program modules defined in analogy to Gaifman & Shapiro (1989), that will be the basis of our work.

2.2.2.1 Formal Preliminaries

Modules, in the sense of Oikarinen & Janhunen (2008), are essentially sets of rules with an input and output interface: A **logic programming module** $\mathcal{P}$ is a tuple $\langle R, I, O, H \rangle$ where: R is a finite set of rules; I, O, and H are pairwise disjoint sets of input, output, and hidden atoms; $At(R) \subseteq At(\mathcal{P})$ defined by $At(\mathcal{P}) = I \cup O \cup H$; and $\forall r \in R:\ Head(r) \notin I$ or, alternatively, $Head(R) \cap I = \emptyset$.

Definition 2.2.1 (Program Module) *A logic program module $\mathcal{P}$ is a tuple $\langle R, I, O, H \rangle$ where:*

1. *R is a finite set of rules;*
2. *I, O, and H are pairwise disjoint sets of input, output, and hidden atoms;*
3. *$At(R) \subseteq At(\mathcal{P})$ defined by $At(\mathcal{P}) = I \cup O \cup H$; and*
4. *$Head(R) \cap I = \emptyset$.*

▲

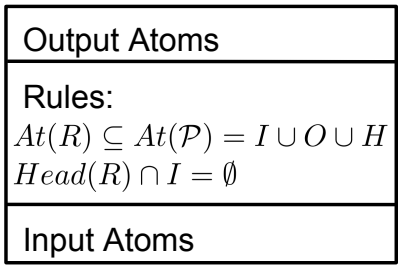


Figure 2.1: Schematic Representation of Module in MLP

The set of atoms in $At_v(\mathcal{P}) = I \cup O$ are considered to be *visible* and hence accessible to other modules composed with $\mathcal{P}$ either to produce input for $\mathcal{P}$ or to make use of the output of $\mathcal{P}$. We use $At_{in}(\mathcal{P}) = I$ and $At_{out}(\mathcal{P}) = O$ to represent the input and output signatures of $\mathcal{P}$, respectively. The hidden atoms in $At_h(\mathcal{P}) = At(\mathcal{P}) \backslash At_v(\mathcal{P}) = H$

are used to formalize some auxiliary concepts of $\mathcal{P}$ which may not be sensible for other modules but may save space substantially. The condition $Head(R) \notin I$ ensures that a module may not interfere with its own input by defining input atoms of I in terms of its rules. Thus, input atoms are only allowed to appear as conditions in rule bodies.

Use case We present next a use case that shows the need for using a modular framework for logic programming, which we depict in Figure 2.2 and will use extensively throughout the rest of the thesis.

Example 2.2.1 *Alice wants to buy a car, wanting it to be safe and not expensive; she preselected three cars, namely c_1, c_2 and c_3. Her friend Bob says that car c_2 is expensive, while Charlie says that car c_3 is expensive. Meanwhile, she consulted two car magazines reviewing all three cars. The first considered c_1 safe and the second considered c_1 to be safe while saying that c_3 may be safe. Alice is very picky regarding safety, and so she seeks some kind of agreement between the reviews.*

The described situation can be captured with five modules, one for Alice, other two for her friends, and two more, one for each magazine. Alice should conclude that c_1 is safe since both magazines agree on this. Therefore, one would expect Alice to opt for car c_1 since it is not expensive, and it is reviewed as being safe. However, some meta-programming techniques can be applied since they share common output atoms. ■

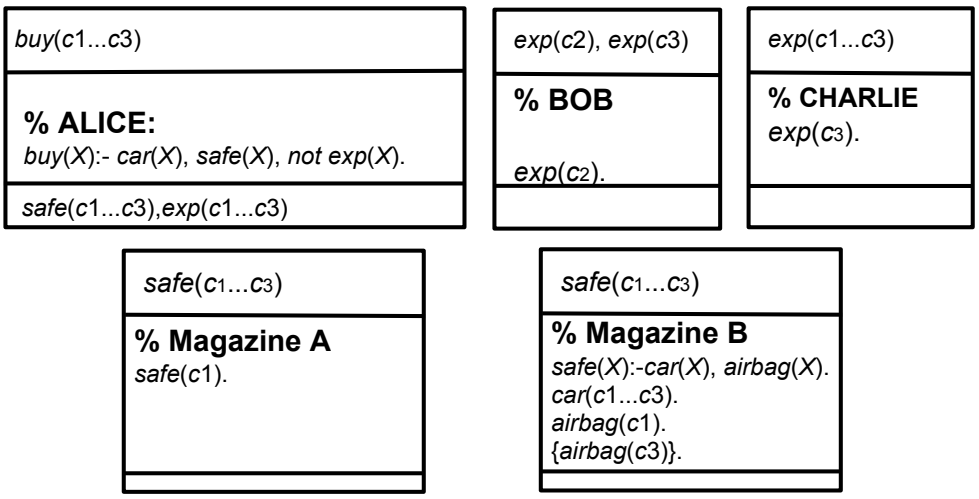


Figure 2.2: Alice Example in MLP's schematic representation.

The use case in Example 2.2.1 and Figure 2.2 is encoded into the five modules shown next in Example 2.2.2:

Example 2.2.2

$$\mathcal{P}_A = \left\langle \begin{array}{ll} R = \left\{ \begin{array}{l} buy(X) \leftarrow car(X), safe(X), not\ exp(X). \\ car(c_1).\quad car(c_2).\quad car(c_3). \end{array} \right\} \\ I = \left\{ \begin{array}{l} safe(c_1), safe(c_2), safe(c_3), \\ exp(c_1), exp(c_2), exp(c_3) \end{array} \right\} \\ O = \{\ buy(c_1), buy(c_2), buy(c_3)\ \} \\ H = \{\ car(c_1), car(c_2), car(c_3)\ \} \end{array} \right\rangle$$

$$\mathcal{P}_B = \left\langle \begin{array}{l} R = \{\ exp(c_2).\}, \\ I = \{\ \}, \\ O = \{\ exp(c_2), exp(c_3)\}, \\ H = \{\ \} \end{array} \right\rangle$$

$$\mathcal{P}_C = \left\langle \begin{array}{l} R = \{\ exp(c_3).\}, \\ I = \{\ \}, \\ O = \{\ exp(c_1), exp(c_2), exp(c_3)\}, \\ H = \{\ \} \end{array} \right\rangle$$

$$\mathcal{P}_{mg_1} = \left\langle \begin{array}{l} R = \{\ safe(c_1).\}, \\ I = \{\ \}, \\ O = \{\ safe(c_1), safe(c_2), safe(c_3)\}, \\ H = \{\ \} \end{array} \right\rangle$$

$$\mathcal{P}_{mg_2} = \left\langle \begin{array}{ll} R = \left\{ \begin{array}{l} safe(X) \leftarrow car(X), airbag(X). \\ car(c_1).\ car(c_2).\ car(c_3). \\ airbag(c_1).\{airbag(c_3)\}. \end{array} \right\}, \\ I = \{\ \}, \\ O = \{\ safe(c_1), safe(c_2), safe(c_3)\ \}, \\ H = \left\{ \begin{array}{l} airbag(c_1), airbag(c_2), airbag(c_3), \\ car(c_1), car(c_2), car(c_3) \end{array} \right\} \end{array} \right\rangle$$

In this Example 2.2.2, module $\mathcal{P}_A$ encodes the rule used by Alice to decide if a car should be bought. The safe and expensive atoms are its inputs, and the buy/1 atoms its outputs; it uses hidden atoms $car/1$ to represent the domain of variables. Modules $\mathcal{P}_B$, $\mathcal{P}_C$ and $\mathcal{P}_{mg_1}$ capture the factual information in Example 2.2.1.

They have no input and no hidden atoms, but Bob has only analyzed the price of cars c_2 and c_3. The ASP program module for the second magazine is more interesting[6]*, and expresses the rule used to determine if a car is safe, namely that a car is safe if it has an airbag; it is known that car c_1 has an airbag, c_2 does not, and the choice rule states that car c_3 may or may not have an airbag.* △

[6] *car* belongs to both hidden signatures of $\mathcal{P}_A$ and $\mathcal{P}_{mg_2}$ which is not allowed when composing these modules, but for clarity we omit a renaming of the $car/1$ predicate.

Semantics of Modular Logic Programs Next, the answer set semantics is generalised to cover modules by introducing a generalisation of the Gelfond-Lifschitz's fixed point definition. In addition to weekly default literals (i.e., *not*), also literals involving input atoms are used in the stability condition. In Oikarinen & Janhunen (2008), the answer sets of a module are defined as follows:

Definition 2.2.2 (Answer Sets of Modules) *An interpretation $M \subseteq At(\mathcal{P})$ is an answer set of an ASP program module $\mathcal{P} = \langle R, I, O, H \rangle$, if and only if:*

$$M = LM\left(R^M \cup \{a. \mid a \in M \cap I\}\right).$$

The answer sets of $\mathcal{P}$ are denoted by $AS(\mathcal{P})$. ▲

Intuitively, the answer sets of a module are obtained from the answer sets of the rules part, for each possible combination of the input atoms.

Example 2.2.3 *Program modules $\mathcal{P}_B$, $\mathcal{P}_C$, and $\mathcal{P}_{mg_1}$ have each a single answer set:*

$$AS(\mathcal{P}_B) = \{\{exp(c_2)\}\}$$

$$AS(\mathcal{P}_C) = \{\{exp(c_3)\}\}$$

and:

$$AS(\mathcal{P}_{mg_1}) = \{\{safe(c_1)\}\}$$

Module $\mathcal{P}_{mg_2}$ has two answer sets, namely:

$$\{safe(c_1), car(c_1), car(c_2), car(c_3), airbag(c_1)\}$$

and:

$$\{safe(c_1), safe(c_3), car(c_1), car(c_2), car(c_3), airbag(c_1), airbag(c_3)\}$$

Alice's ASP program module has $2^6 = 64$ models, each corresponding to an input combination of safe and expensive atoms. Some of these models are:

$$\begin{array}{ll} \{ & buy(c_1), car(c_1), car(c_2), car(c_3), safe(c_1) \quad \} \\ \{ & buy(c_1), buy(c_3), car(c_1), car(c_2), car(c_3), \\ & safe(c_1), safe(c_3) \quad \} \\ \{ & buy(c_1), car(c_1), car(c_2), car(c_3), exp(c_3), \\ & safe(c_1), safe(c_3) \quad \} \end{array}$$

■

Composing programs from modules The composition of modules is obtained from the union of program rules and by constructing the composed output set as the union of the output sets of the modules, thus removing from the input all the specified output atoms. Oikarinen & Janhunen (2008) define their first composition operator as follows:

Definition 2.2.3 *Given two modules* $\mathcal{P}_1 = \langle R_1, I_1, O_1, H_1 \rangle$ *and* $\mathcal{P}_2 = \langle R_2, I_2, O_2, H_2 \rangle$*, their composition* $\mathcal{P}_1 \oplus \mathcal{P}_2$ *is defined when their output signatures are disjoint, that is,* $O_1 \cap O_2 = \emptyset$*, and they respect each others hidden atoms, i.e.,* $H_1 \cap At(\mathcal{P}_2) = \emptyset$ *and* $H_2 \cap At(\mathcal{P}_1) = \emptyset$*. Then their composition is*

$$\mathcal{P}_1 \oplus \mathcal{P}_2 = \langle R_1 \cup R_2, (I_1 \setminus O_2) \cup (I_2 \setminus O_1), O_1 \cup O_2, H_1 \cup H_2 \rangle.$$

▲

However, the conditions given for $\oplus$ are not enough to guarantee compositionality in the case of answer sets and as such they define a restricted form:

Definition 2.2.4 (Module Union Operator $\sqcup$) *Given modules* $\mathcal{P}_1, \mathcal{P}_2$*, their union is* $\mathcal{P}_1 \sqcup \mathcal{P}_2 = \mathcal{P}_1 \oplus \mathcal{P}_2$ *whenever:*

(i) $\mathcal{P}_1 \oplus \mathcal{P}_2$ *is defined and*

(ii) $\mathcal{P}_1$ *and* $\mathcal{P}_2$ *are mutually independent* [7]. ▲

Natural join ($\bowtie$) on visible atoms is used in Oikarinen & Janhunen (2008) to combine answer sets of modules as follows:

Definition 2.2.5 (Join) *Given modules* $\mathcal{P}_1$ *and* $\mathcal{P}_2$ *and sets of interpretations* $A_1 \subseteq 2^{At(\mathcal{P}_1)}$ *and* $A_2 \subseteq 2^{At(\mathcal{P}_2)}$*, the natural join of* A_1 *and* A_2 *is:*

$$A_1 \bowtie A_2 = \left\{ \begin{array}{l} M_1 \cup M_2 \mid M_1 \in A_1, M_2 \in A_2 \text{ and} \\ M_1 \cap At_v(\mathcal{P}_2) = M_2 \cap At_v(\mathcal{P}_1) \end{array} \right\}.$$

▲

This leads to their main result:

Theorem 2.2.1 (Module Theorem in Oikarinen & Janhunen (2008)) *If* $\mathcal{P}_1, \mathcal{P}_2$ *are modules such that* $\mathcal{P}_1 \sqcup \mathcal{P}_2$ *is defined, then*

$$AS(\mathcal{P}_1 \sqcup \mathcal{P}_2) = AS(\mathcal{P}_1) \bowtie AS(\mathcal{P}_2).$$

○

Still according to Oikarinen & Janhunen (2008), their module theorem also straightforwardly generalises for a collection of modules because the module union operator $\sqcup$ is commutative, associative, and has the identity element $< \emptyset, \emptyset, \emptyset, \emptyset >$.

[7] There are no positive cyclic dependencies among rules in different modules, defined as loops through input and output signatures.

Example 2.2.4 *Consider the composition $\mathcal{Q} = (\mathcal{P}_A \sqcup \mathcal{P}_{mg_1}) \sqcup \mathcal{P}_B$. First, we have*

$$\mathcal{P}_A \sqcup \mathcal{P}_{mg_1} = \left\langle \begin{array}{l} \{buy(X) \leftarrow car(X), safe(X), exp(X). \\ \ car(c_1).\ car(c_2).\ car(c_3).\ safe(c_1).\}, \\ \{exp(c_1), exp(c_2), exp(c_3)\}, \\ \{buy(c_1), buy(c_2), buy(c_3), \\ \ safe(c_1),\ safe(c_2),\ safe(c_3)\}, \\ \{car(c_1), car(c_2), car(c_3)\} \end{array} \right\rangle$$

It is immediate to see that the module theorem holds in this case. The visible atoms of $\mathcal{P}_A$ are $safe/1$, $exp/1$ and $buy/1$, and the visible atoms for $\mathcal{P}_{mg_1}$ are $\{safe(c_1), safe(c_2)\}$. The only model for $\mathcal{P}_{mg_1} = \{safe(c_1)\}$ when naturally joined with the models of $\mathcal{P}_A$, results in eight possible models where $safe(c_1)$, not $safe(c_2)$, and not $safe(c_3)$ hold, and the $exp/1$ atoms vary. The final ASP program module $\mathcal{Q}$ is

$$\left\langle \begin{array}{ll} \{ & buy(X) \leftarrow car(X), safe(X), not\ exp(X). \\ & car(c_1).\ car(c_2).\ car(c_3).\ exp(c_2).\ safe(c_1).\}, \\ \{ & exp(c_1)\}, \\ \{ & buy(c_1), buy(c_2), buy(c_3), exp(c_2), exp(c_3), safe(c_1), safe(c_2), safe(c_3)\}, \\ \{ & car(c_1), car(c_2), car(c_3)\} \end{array} \right\rangle$$

The answer sets of $\mathcal{Q}$ are thus:

$$\{safe(c_1), exp(c_1), exp(c_2), car(c_1), car(c_2), car(c_3)\}$$
$$\{buy(c_1), safe(c_1), exp(c_2), car(c_1), car(c_2), car(c_3)\}$$

■

2.2.2.2 Visible and Modular Equivalence

The notion of visible equivalence has been introduced in order to neglect hidden atoms when logic programs are compared on the basis of their models. The compositionality property from the module theorem enabled the authors to port this idea to the level of program modules — giving rise to modular equivalence of logic programs.

Definition 2.2.6 (Visible and Modular Equivalence) *Given two logic program modules $\mathcal{P}$ and $\mathcal{Q}$, they are:*

Visibly equivalent: *$\mathcal{P} \equiv_v \mathcal{Q}$ iff $At_v(\mathcal{P}) = At_v(\mathcal{Q})$ and there is a bijection $f : AS(\mathcal{P}) \rightarrow AS(\mathcal{Q})$ such that for all $M \in AS(\mathcal{P})$, $M \cap At_v(\mathcal{P}) = f(M) \cap At_v(\mathcal{Q})$.*

Modularly equivalent: *$\mathcal{P} \equiv_m \mathcal{Q}$ iff $At_{in}(\mathcal{P}) = At_{in}(\mathcal{Q})$ and $\mathcal{P} \equiv_v Q$.*

▲

So, two modules are visibly equivalent if there is a bijection among their answer sets, and they coincide in their visible parts. If additionally, the two program modules have the same input and output atoms, then they are modularly equivalent.

2.2.2.3 Shortcomings of Modular Logic Programming

The conditions imposed in these definitions bring about some shortcomings such as the fact that the output signatures of two modules must be disjoint which disallows many practical applications.

Alice wanting to buy a car can be captured with five modules, three for Alice and her friends, and two more for magazines A and B. Alice should conclude that car 1 is safe since both magazines agree on this. Therefore, one would expect Alice to opt for car 1 since it is not expensive, and it is reviewed as being safe. However, the current state-of-the-art does not provide any way of combining these modules since they share common output atoms.

Example 2.2.5 *In Figure 2.2, one is not able to combine the results of program modules for magazines A and B ($\mathcal{P}_{MagA}$ and $\mathcal{P}_{MagB}$), and thus it is impossible to obtain the combination of the five modules. Also because of this, the module union operator $\sqcup$ is not reflexive in general. By trivially waiving this condition, we immediately get problems with conflicting modules.* △

Example 2.2.6 (Common Outputs) *Given modules $\mathcal{P}_B$ and $\mathcal{P}_C$(for Bob and Charlie respectively), which respectively have $AS(\mathcal{P}_B) = \{\{exp(c_2)\}\}$ and $AS(\mathcal{P}_C) = \{\{exp(c_3)\}\}$, we have that $AS(\mathcal{P}_B \sqcup \mathcal{P}_C) = \{\{exp(c_2), exp(c_3)\}\}$. However,*

$$AS(\mathcal{P}_B) \bowtie AS(\mathcal{P}_C) = \emptyset$$

invalidating the module theorem. ■

The compatibility criterion for the operator $\bowtie$ rules out the compositionality of mutually dependent modules while allowing positive loops inside individual modules and negative loops in general. Consider now the following example:

Example 2.2.7 (Cyclic Dependencies) *Given the following programs:*

$$\mathcal{P}_1 = \langle \{airbag \leftarrow safe.\}, \{safe\}, \{airbag\}, \emptyset \rangle$$

$$\mathcal{P}_2 = \langle \{safe \leftarrow airbag.\}, \{airbag\}, \{safe\}, \emptyset \rangle$$

which respectively have the following answer sets: $AS(\mathcal{P}_1) = \{\{\}, \{airbag, safe\}\}$ and $AS(\mathcal{P}_2) = \{\{\}, \{airbag, safe\}\}$ while $AS(\mathcal{P}_1 \oplus \mathcal{P}_2) = \{\{\}\}$.

Therefore,

$$AS(\mathcal{P}_1 \oplus \mathcal{P}_2) \neq AS(\mathcal{P}_1) \bowtie AS(\mathcal{P}_2) = \{\{\}, \{airbag, safe\}\}$$

and the module theorem is not applicable in this setting. ■

2.3 Conclusions

In this chapter we have overviewed the most important formalisms related to logic programming and modular logic programming, namely the syntax and semantics of answer set programs, an overview of the logic of here-and-there and of equilibrium logic and introduced the notions of paraconsistent and paracoherent reasoning.

We leave, however, particular details that are only locally relevant to be introduced in chapters ahead. As an example of such details, the background and relevant formalisms in the context of probabilistic logic programming are introduced in Chapter 6, Multi Context Systems are discussed in depth in Chapter 5, while details related to provenance propositional formulas for logic programs Viegas Damásio *et al.* (2013) are introduced in Chapter 8. Also, default logic and the concepts of strong and relativised equivalence of logic programs are formally introduced in Chapter 9.

Chapter 3

Outline of the Thesis

This thesis manuscript is split into four parts: two technical ones plus the current part (Part I) for providing context and introduction, and the last part (Part IV) for conclusions. The current chapter serves as a summary of the thesis and can be read alone if necessary. Those interested in modularity should read the second part (Part II) while those interested in Conflicts in Logic Programming should read the third part (Part III) after optionally reading the first two chapters of the second part. In addition, this thesis is written in a way such that each chapter can be read individually in as much as they are self-contained regarding the discussion of results and suggestions for future work albeit we introduce most of the concepts and formalisms in the first part and in each chapter we then introduce only the concepts whose use is limited in scope to that respective chapter. Of course understanding the discussions we include about the way in which different chapters are related will only be possible by reading all relevant chapters.

3.1 Part II: How can we generalise and extend Modular in Logic Programming?

In summary, the fundamental results of Oikarinen & Janhunen (2008) require a syntactic operation to combine modules (basically the union of programs), and a semantic operation joining the models of the modules. The module theorem states that the models of the combined modules can be obtained by applying the semantic join operation to the original models of the modules. The semantics is compositional but, as said before, suffers from two problems in terms of applicability.

3.1.1 Chapter 4: Allowing Common Outputs

Example 2.2.6 shows that allowing common outputs destroys the property of compositionality. We pursued two alternatives: the first is to keep the original syntactic operation which implies using the union of programs to syntactically combine modules, plus adding some bookkeeping of the interface. In this case the semantic operation

on models has to be changed; the second alternative is to keep the original semantic join operation for models, and because of that a new syntactic operation is required to guarantee compositionality.

Damásio & Moura (2014) showed that keeping the syntactic operation (the first solution) is impossible and in this chapter we present a solution to this problem based on a renaming transformation that introduces the required extra information. The second solution is possible, and builds on the previous modular transformation.

3.1.2 Chapter 5: Allowing Cyclic Dependencies Between Modules

Example 2.2.7 shows that positive loops between modules also destroy the property of compositionality. This chapter extends and improves preliminary work in Damásio & Moura (2015) which discussed a solution towards solving this problem.

We present in this chapter a model join operation that requires one to look at every model of two modules being composed in order to check for minimality of models that are comparable on account of their inputs. This operation is able to distinguish between atoms that are self-supported through positive loops and atoms with proper support, allowing one to lift the condition disallowing positive dependencies between modules. However, this approach is not local as it requires comparing all models and, as it is not general because it does not allow combining modules with default negation, it is of limited applicability.

Because of the lack of generality of the former approach, we present an alternative solution requiring the introduction of extra information in the models for one to be able to detect dependencies. We use models annotated with the way they depend on the atoms in the input signature of their module. We then define their semantics in terms of a fixed point operator. The original composition operators are applicable to annotated modules if positive dependencies for atoms are added to their respective models. This approach turns out to be local, in the sense that we need only look at two models being joined and unlike the first alternative we presented, it works well with normal logic programs.

3.1.3 Chapter 6: A Modular Extension of Probabilistic Logic Programming

The P-log language Baral *et al.* (2004) has emerged as one of the most flexible frameworks for combining probabilistic reasoning with logical reasoning, in particular by distinguishing acting (doing) from observations and allowing non-trivial conditioning forms Baral *et al.* (2004); Baral & Hunsaker (2007). The P-log language is a non-monotonic probabilistic logic language supported on two major formalisms, namely Gelfond & Lifschitz (1988) for declarative knowledge representation and Causal Bayesian Networks Pearl (1988, 2000) (CBNs) as its probabilistic foundation.

For answering probabilistic logic programming queries we pursued a method that does not imply calculating all answer sets (and neither computing complete ones) for a given program. and in Damásio & Moura (2011) provided a semantics that enables

this modularisation for P-Log. Furthermore the calculations of answer sets that are subsumed by others already made for a given query can be reused and thus, the property of compositionality is maintained. It is the first approach in the literature that modularises P-log programs and is able to make their composition incrementally by combining compatible possible worlds and multiplying corresponding unnormalised conditional probability measures. We do so by resorting to MLP Oikarinen & Janhunen (2008) and by eliminating variables in modules which reduces the space and time necessary to make inference, whereas previous algorithms require enumeration of all possible worlds (exponential on the number of random variables) and repeat calculations. We optimised the case of single connected Bayesian Networks (polytrees), in which there is only one path between any two node, and perform reasoning in polynomial time but as expected, the general case of exact inference is intractable. One must still consider methods for approximate inference (e.g., by extending sampling algorithms) which are outside the scope of this thesis though.

We fully describe the inference algorithm obtained from the compositional semantics of P-log modules and related it formally with the variable elimination algorithm. We are also able to handle probabilistic conditioning (observations) and actions.

3.2 Part III: How Can We Deal With Conflicts in Logic Programming?

When combining LPs from different origins, conflicts may occur. There are, in general, two ways to solve them, again: **(1) Semantically**, by using paraconsistent/paracoherent semantics; **(2) Syntactically**, fixing those problems by suggesting changes to the program modules, by means justifications and debugging models.

3.2.1 Chapter 7: Paracoherent Reasoning

Our work in Eiter *et al.* (2010b); Amendola *et al.* (2015) focused on contributing to a more logical foundation of paracoherent answer set programming, which gains increasing importance in inconsistency management. Some interesting and motivating topics involve e.g., using paracoherent semantics in diagnosis (exploit assumptions to generate diagnoses) or extensions to modular logic programs, where module interaction may lead to incoherence.

We studied the problem of reasoning from incoherent answer set programs, i.e., from logic programs that do not have an answer set due to cyclic dependencies of an atom from its default negation. As a starting point we considered so-called semi-stable models which have been developed for this purpose building on a program transformation, called epistemic transformation. We then provided a model-theoretic characterisation of this semantics, considering pairs of two-valued interpretations (similar to here-and-there models) of the original program, rather than resorting to its epistemic transformation. This allowed us to show some anomalies of semi-stable

semantics with respect to basic epistemic properties and lead us to propose an alternative semantics satisfying these properties. In addition to a model-theoretic and a transformational characterisation of the alternative semantics, we have proven precise complexity results for major reasoning tasks under both semantics.

In summary, we have addressed the problematic of reasoning from incoherent knowledge bases and make the following main contributions.

- We characterise semi-stable models by pairs of 2-valued interpretations of the original program, similar to so-called here-and-there (HT) models in equilibrium logic Pearce (2006b); Pearce & Valverde (2008). In the course of this, we point out some anomalies of the semi-stable semantics with respect to basic rationality properties in modal logics (**K** and **N**) which essentially prohibit a 1-to-1 characterisation[1] in terms of HT-models. Roughly speaking, the epistemic transformation misses some links between atoms encoding truth values of atoms, which may lead in some cases to unintuitive results.

- The anomalies lead us to propose an alternative paracoherent semantics, called *semi-equilibrium ($\mathcal{SEQ}$) model semantics*, which remedies the anomalies of the semi-stable model semantics. It satisfies the properties (D1)-(D3) presented in Section 2.1.3 and is fully characterised using HT-models. Informally, semi-equilibrium models are 3-valued interpretations in which atoms can be true, false or believed true; the gap between believed and (derivably) true atoms is globally minimised. Note that the semantic distinction between believed true and true atoms in models is important. Other approaches, e.g. CR-Prolog Balduccini & Gelfond (2003), make a syntactic distinction at the rule level which does not semantically discriminate believed atoms; this may lead to more models. Notably, $\mathcal{SEQ}$-models can be obtained by an extension of the epistemic transformation that adds further rules.

3.2.2 Chapter 8: Unifying Justifications and Debugging for Answer Set Programming

Most of the work in the literature has been concerned with the problem of debugging, neglecting that a notion of provenance could provide better explanations for the problems found. The approaches to debugging of answer sets either use complex graph constructs Pontelli *et al.* (2009), or resort to meta-programming approaches Gebser *et al.* (2008). The use of meta-programming approaches to perform debugging is not novel (see for instance Pereira *et al.* (1993a)), but current techniques of Gebser *et al.* (2008) are more informative. However, determining provenance models has been conjectured as a viable complementary technique for debugging Viegas Damásio *et al.* (2013).

[1] By 1-to-1 we mean a one to one and onto (i.e., bijective) correspondence.

Our program transformation in Viegas Damásio *et al.* (2013) allows the computation of why-not provenance models under the well-founded and stable model semantics. We do this in a modular way trying to keep compatibility with the previous work of Viegas Damásio *et al.* (2013). This enables the computation of provenance answer sets in an easy way by using answer set solvers. Having this, we align provenance answer sets with the ones from debugging in a single unified transformation and show that in fact the provenance approach in some sense generalises the debugging approach, since any error has a counterpart provenance but not the other way around.

Since the proposed method is based on meta-programming, it is possible to use existing state-of-the-art ASP solvers that support well-founded and answer set semantics, which allowed us to develop a new prototypical tool[2] by extending the one that exists related to the debugging approach we followed: spock Gebser *et al.* (2007b). We explore this in Damásio *et al.* (2015), which unifies provenance and debugging under a common framework and term these explanations 'justifications'.

3.2.2.1 Debugging

Debugging of logic programs and in particular ASP has received important contributions over the last years. For instance Eiter *et al.* (2010a) provided two approaches for explaining inconsistency, both of which characterise inconsistency in terms of bridge rules, but in different ways: by pointing out rules which need to be altered for restoring consistency, and by finding combinations of rules which cause inconsistency.

We are mostly interested in the approach by Gebser *et al.* (2008) though, where a meta-programming technique for debugging ASP is presented. Debugging queries are expressed by answer set programs, which allows for restricting debugging information to relevant parts. The basic question addressed is why interpretations expected to be answer sets are not answer sets of a given program, thus it finds semantic errors in programs. The explanations provided are based on a scheme of errors that relies on loop-formulae, an alternative characterisation of the answer set semantics Lee (2005); Ferraris *et al.* (2007), and encompasses four different causes, namely: **Unsatisfied rules**, **Violated integrity constraints**, **Unsupported atoms** and **Unfounded loops**. A meta-program is constructed from a given program Π and an interpretation I that is capable to detect the above four errors via occurrences of the special atoms in its answer sets reflecting the errors. The predicates of these meta-atoms are called error-indicating predicates.

In recent practical work on this subject, Shchekotykhin (2014) presented an interactive query-based ASP debugging method that finds an explanation by means of observations inputed by some user that reflect what he/she believes to be a preferred explanation. The system queries a programmer whether a set of ground atoms must be true in all (cautious reasoning) or some (brave reasoning) answer sets of the program.

However, these approaches do not answer the question of why a given interpretation is indeed an answer set and hence do not provide justifications for answer sets.

[2] Available for download at https://sourceforge.net/projects/cptkirk/

3.2.3 Chapter 9: Real World Application Field: Access Control Policies

For a long time now, logic programming and rule-based reasoning have been proposed as a strong basis for policy specification languages. However, the term "policy" has never been given a unique meaning. In fact, it is used in the literature in an ambiguous and broad sense that encompasses at least the following notions: **Access Control Policies** (ACP) are policies that pose constraints on the behaviour of a system. They are typically used to control permissions of users/groups accessing resources and services;

In Kolovski (2007); Bonatti *et al.* (2009), one can find good introductory surveys to logic-based ACPs. Kolovski (2007) is limited to the presentation of a not formally characterised DL-based formalism to represent policies, while Bonatti *et al.* (2009) considers a general overview.

3.2.3.1 Hospital Access Control Policy in Modular Logic Programming

Recall now the hospital example we used earlier which is captured and encoded into a framework presented in Bonatti *et al.* (2002). However, this framework is not compositional in the sense we are interested in: one cannot simply take the models of individual modules and combine them by using some join operator in order to obtain the models of the composed modules but rather has to union the modules and then calculate the union's models. We now take Examples 1.2.1, 1.2.2, and 1.2.3 and present an adapted translation into MLP. Consider that every module has predicates $patient(P)$ and $documents(D)$ in its input signature. We omit policies for the radiology department as well as policies P_{reg} and P_{trial}.

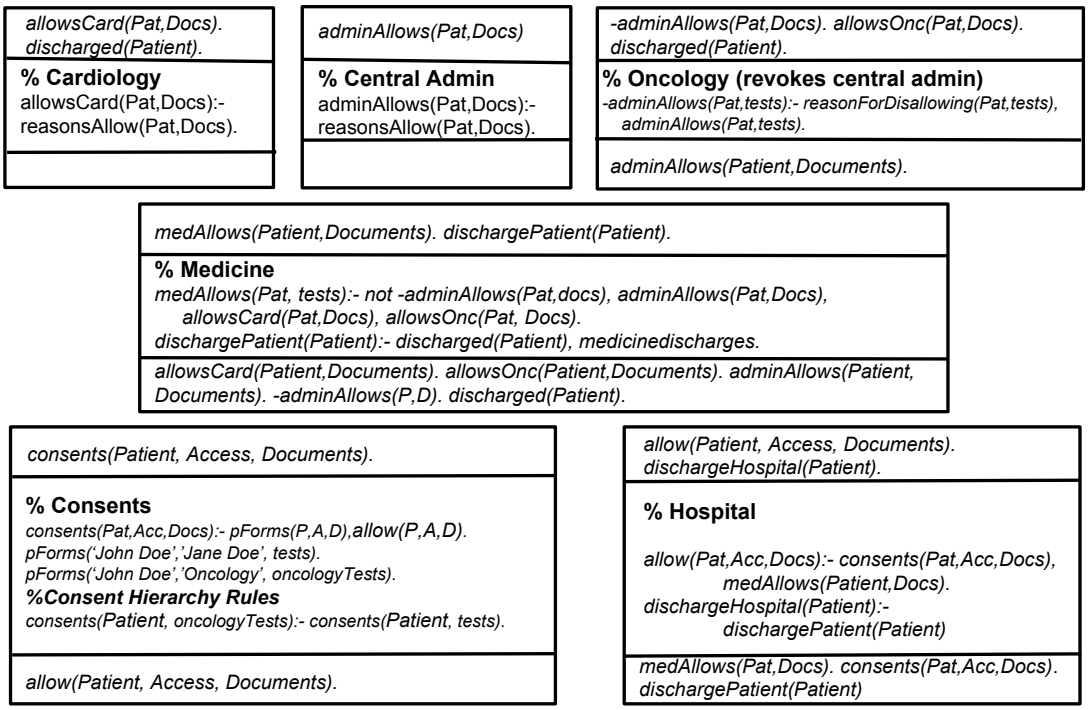


Figure 3.1: Hospital Policy

In Moura (2012) we formally characterised conflicts in ACPs in terms of a meta-model by Barker (2009), as:

Modality Conflicts are inconsistencies in the policy specification that may arise when two or more policies with opposite modalities refer to the same authorisation subjects, actions and objects – e.g., in ASP having both allow and deny predicates in the same model in presence of an integrity constraint disallowing so.

Redundancy Conflicts: An ACP is redundant if a permission or a prohibition can be derived from another set of applicable ACPs. Though redundancy conflict have no influence in the enforcement of ACPs, they should be identified and dealt with because a redundant policy often reflects an error that was made while describing security requirements.

Potential Conflicts: Potential conflicts between two policies having overlaps in the expression of their conditions exist if it is the case that there are no modality or redundancy conflicts between the rules in two polices, but when the conditions of such rules are simultaneously met, the two policies can result in modality or redundancy conflicts. According to the definition, when some policies have the same condition literals, we can infer the existence of potential conflicts among

these policies. Consequently, potential conflicts are highly pervasive in access control systems.

3.2.4 Chapter 10: Conclusions and Future Work

In the final chapter of this thesis we present overall conclusions, identify gaps and point directions for future work.

3.3 Conclusions

In this Part I we summarised our work and set the goals of this thesis, namely, to provide the theoretical foundations for a generalised modular answer set programming framework. This framework generalises the Modular Logic Programming results by Oikarinen & Janhunen (2008), providing additional operators for combining arbitrary logic programming modules while preserving the compositional nature of MLP. We provide characterisations of conflicts that occur when composing these generalised modules and we are able to, by unifying provenance and debugging for LP modules, effectively give why and why-not justifications for answer sets. We complement our work with computational complexity results (for the work in Chapter 7) and and provide a prototypical tool (for the work in Chapter 8).

We cover all of these results with published work and motivate a future link of these theoretical contributions and make use of a prototypical tool implementing parts of this framework, in the context of access control policies, as a practical real-world scenario.

We are also aware of current gaps in our work and identify interesting directions for future work throughout this Thesis and particularly in the last chapter (Chapter 10).

Part II

Generalising Modular Logic Programming

Part Overview:

We depart from the Modular Logic Programming (MLP) approach in Gaifman & Shapiro (1989); Oikarinen & Janhunen (2008) which achieved a restricted form of compositionality of ASP modules. Our goal is to generalise MLP and we start by lifting the restrictive conditions that were originally imposed, namely disallowing common outputs and positive cyclic dependencies, presenting alternative ways for combining these modules and thus develop a fully compositional framework: Generalised Modular Logic Programming (GMLP).

In Chapter 4 we redefine the necessary operators in order to relax the conditions for combining modules with common atoms in their output signatures. Two alternative solutions are presented, both allowing us to retain compositionality while dealing with a more general setting than before.

In Chapter 5 we lift the restriction that disallows composing modules with cyclic dependencies in the framework of Modular Logic Programming Oikarinen & Janhunen (2008). We also present two alternatives solutions for this: a model join operation that requires one to look at every model of two modules being composed in order to check for minimality of models that are comparable on account of their inputs; an alternative solution requiring the introduction of extra information in the models for one to be able to detect dependencies. In this approach we use models annotated with the way they depend on the atoms in their module's input signature and show that this is local in the sense that we only need to look at the two models being joined and unlike the first alternative we presented, it works well with integrity constraints. We provide a comparison between MLP and MCS in terms of their suitability for compositionality in the sense of MLP's module theorem and our definition of compositionality.

In Chapter 6 we present the first approach in the literature to modularise P-log programs and to make their composition incrementally by combining compatible possible worlds and multiplying corresponding unnormalised conditional probability measures. As expected, it turns out that the general case of exact inference is intractable but for the case of Bayesian Networks with a polytree structure represented in P-log, we can do reasoning with P-log in polynomial time.

Chapter 4

Allowing Common Outputs Between Modules

Even though modularity has been studied extensively in conventional logic programming, there are few approaches on how to incorporate modularity into Answer Set Programming. A major approach is Oikarinen and Janhunen's Gaifman-Shapiro-style architecture of program modules Gaifman & Shapiro (1989); Oikarinen & Janhunen (2008), which provides the composition of program modules. Their module theorem properly strengthens Lifschitz and Turner's splitting set theorem for normal logic programs Lifschitz & Turner (1994). However, this approach is limited by module conditions that are imposed in order to ensure the compatibility of their module system with the answer set semantics, namely forcing output signatures of composing modules to be disjoint and disallowing positive cyclic dependencies between different modules. These conditions turn out to be too restrictive in practice and in this chapter we discuss alternative ways of effectively lifting the first restriction, while leaving the second restriction to be dealt with in the next chapter, widening the applicability of this framework and the scope of the module theorem.

4.1 Introduction

Over the last few years, answer set programming (ASP) Eiter *et al.* (2001); Baral (2003); Lifschitz (2002); Marek & Truszczynski (1999); Niemelä (1998) emerged as one of the most important methods for declarative knowledge representation and reasoning. Despite its declarative nature, developing ASP programs resembles conventional programming: one often writes a series of gradually improving programs for solving a particular problem, e.g., optimising execution time and space. Until recently, ASP programs were considered as integral entities, which becomes problematic as programs become more complex, and their instances grow.

Even though modularity is extensively studied in logic programming Gaifman & Shapiro (1989), there are only a few approaches on how to incorporate it into

ASP Oikarinen & Janhunen (2008); Dao-Tran *et al.* (2009); Babb & Lee (2012) or other module-based constraint modeling frameworks Järvisalo *et al.* (2009); Tasharrofi & Ternovska (2011). The research on modular systems of logic programming has followed two main-streams Bugliesi *et al.* (1994). One is programming in-the-large where compositional operators are defined in order to combine different modules, e.g., Mancarella & Pedreschi (1988); Gaifman & Shapiro (1989); O'Keefe (1985). These operators allow combining programs algebraically, which does not require an extension of the theory of logic programs. The other direction is programming-in-the-small, e.g., Giordano & Martelli (1994); Miller (1986), aiming at enhancing logic programming with scoping and abstraction mechanisms available in other programming paradigms. This approach requires the introduction of new logical connectives in an extended logical language. The two mainstreams are thus quite divergent.

The approach of Oikarinen & Janhunen (2008) defines modules as structures specified by a program (knowledge rules) and by an interface defined by input and output atoms which for a single module are, naturally, disjoint. The authors also provide a module theorem capturing the compositionality of their module composition operator but impose the two aforementioned conditions. The techniques used in Dao-Tran *et al.* (2009) for handling positive cycles among modules are shown not to be adaptable for the setting of Oikarinen & Janhunen (2008).

In this chapter we discuss two alternative solutions to the common outputs problem, generalising the module theorem by allowing common output atoms in the interfaces of the modules being composed.

In summary, the fundamental results of Oikarinen & Janhunen (2008) require a syntactic operation to combine modules – basically corresponding to the union of programs –, and a compositional semantic operation joining the models of the modules. The module theorem states that the models of the combined modules can be obtained by applying the semantics of the natural join operation to the original models of the modules – which is compositional according to our informal definition, which we recall next:

Compositionality (informally): The combination of models of individual modules are the models of the union of modules.

The authors show however that allowing common outputs destroys this property. There are two alternatives to pursue:

(1) Keep the syntactic operation: use the union of programs to syntactically combine modules, plus some bookkeeping of the interface, and thus the semantic operation on models has to be changed;

(2) Keep the semantic operation: the semantic operation is the natural join of models, and thus a new syntactic operation is required to guarantee compositionality.

Both will be explored in this chapter as they correspond to different and sensible ways of combining two sources of information, already identified in Example 2.2.1: the first alternative is necessary for Alice to determine if a car is expensive; the second alternative captures the way Alice determines whether a car is safe or not. Keeping

the syntactic operation is shown to be impossible since models do not convey enough information to obtain compositionality. We present a solution to this problem based on a transformation that introduces the required extra information. The second solution is possible, and builds on the previous module transformation.

This chapter relies on the overview of the modular logic programming paradigm presented in Section 2.2.2 and we recall next in Section 4.2 its shortcomings. In Section 4.3 introduce two new forms of composing modules allowing common outputs, one keeping the original syntactic *union* operator and the other keeping the original semantic model *join* operator. We finish with conclusions and a general discussion.

4.2 Modularity in Answer Set Programming

Modular aspects of Answer Set Programming have been clarified in recent years, with authors describing how and when two program parts (modules) can be composed Oikarinen & Janhunen (2008); Dao-Tran *et al.* (2009); Järvisalo *et al.* (2009) under the answer set semantics. In this chapter, we make use of Oikarinen and Janhunen's logic program modules defined in analogy to Gaifman & Shapiro (1989) which we extensively reviewed in Section 2.2.2.

4.2.1 Shortcomings

The conditions imposed in these definitions bring about some shortcomings such as the fact that the output signatures of two modules must be disjoint which disallows many practical applications e.g., we are not able to combine the results of program module $\mathcal{Q}$ with any of $\mathcal{P}_C$ or $\mathcal{P}_{mg_2}$ from Example 2.2.2, and thus it is impossible to obtain the combination of the five modules. Also because of this, the module union operator $\sqcup$ is not reflexive. By trivially waiving this condition, we immediately get problems with conflicting modules as shown in Example 4.2.1.

Example 4.2.1 (Common Outputs) *Given modules $\mathcal{P}_B$ and $\mathcal{P}_C$ from Example 2.2.2, which respectively have the following answer sets:*

$$AS(\mathcal{P}_B) = \{\{exp(c_2)\}\} \text{ and } AS(\mathcal{P}_C) = \{\{exp(c_3)\}\}$$

The single answer set of their union $AS(\mathcal{P}_B \sqcup \mathcal{P}_C)$ is:

$$\{exp(c_2), exp(c_3)\}$$

However, the join of their answer sets is

$$AS(\mathcal{P}_B) \bowtie AS(\mathcal{P}_C) = \emptyset$$

invalidating the module theorem. ■

The compatibility criterion for the operator $\bowtie$ also rules out the compositionality of mutually dependent modules, but allows positive loops inside modules or negative loops in general. We will deal with that problem in Chapter 5 and focus next on allowing common output atoms in the signatures of modules under composition.

4.3 Generalising Modularity in ASP by Allowing Common Outputs

After having identified the shortcomings in the literature, we proceed now to seeing how compositionality can be maintained while allowing modules to have common output atoms. In this section we present two versions of compositions:

1. A relaxed composition operator ($\uplus$), aiming at maximising information in the answer sets of modules. Unfortunately, we show that this operation is not compositional.

2. A conservative composition operator ($\otimes$), aiming at maximising compatibility of atoms in the answer sets of modules. This version implies redefining the composition operator by resorting to a program transformation but uses the original join operator.

First, one requires fundamental operations for renaming atoms in the output signatures of modules with fresh ones:

Definition 4.3.1 (Output renaming) *Let $\mathcal{P}$ be the program module $\mathcal{P} = \langle R, I, O, H \rangle$, $o \in O$ and $o' \notin At(\mathcal{P})$. The renamed output program module is:*

$$\rho_{o' \leftarrow o}(\mathcal{P}) = \langle R' \cup \{\bot \leftarrow o', not\ o.\}, I \cup \{o\}, \{o'\} \cup (O \setminus \{o\}), H \rangle \tag{4.1}$$

The program part R' is constructed by substituting the head of each rule $o \leftarrow Body$ in R by $o' \leftarrow Body$. The heads of other rules remain unchanged, as well as the bodies of all rules. ▲

Mark that, by making o an input atom, the renaming operation can introduce extra answer sets. However, the original answer sets can be recovered by selecting the models where o' has exactly the same truth-value as o. The constraint throws away models where o' holds but not o. We will abuse notation and denote $\rho_{o'_1 \leftarrow o_1}\left(\ldots\left(\rho_{o'_n \leftarrow o_n}(\mathcal{P})\right)\ldots\right)$ by $\rho_{\{o'_1,\ldots,o'_n\} \leftarrow \{o_1,\ldots,o_n\}}(\mathcal{P})$. Strictly speaking one should use $(o'_1,\ldots,o'_n) \leftarrow (o_1,\ldots,o_n)$ sentences instead of sets but that would introduce extra notation and significant overhead in forthcoming definitions and results.

Example 4.3.1 (Renaming) *Recall the module representing Alice's conditions in Example 2.2.2.*

$$\mathcal{P}_A = \left\langle \begin{array}{l} \{\ buy(X) \leftarrow car(X), safe(X), not\ exp(X). \\ \quad car(c_1).\quad car(c_2).\quad car(c_3).\}, \\ \{safe(c_1), safe(c_2), safe(c_3),\ exp(c_1), exp(c_2), exp(c_3)\}, \\ \{buy(c_1), buy(c_2), buy(c_3)\}, \{car(c_1), car(c_2), car(c_3)\} \end{array} \right\rangle$$

Its renamed output program module $\rho_{o' \leftarrow o}(\mathcal{P}_A)$ is the program module:

$$\rho_{o' \leftarrow o}(\mathcal{P}_A) = \left\langle \begin{array}{l} \{buy'(X) \leftarrow car(X), safe(X),\ not\ exp(X). \\ \quad car(c_1).\quad car(c_2).\quad car(c_3). \\ \bot \leftarrow buy'(X), not\ buy(X).\}, \\ \{buy(c_1), buy(c_2), buy(c_3), safe(c_1), safe(c_2), safe(c_3), \\ \quad exp(c_1), exp(c_2), exp(c_3)\}, \\ \{buy'(c_1), buy'(c_2), buy'(c_3)\}, \\ \{car(c_1), car(c_2), car(c_3)\} \end{array} \right\rangle$$

Its models are:

$\{exp(c_1), exp(c_2), exp(c_3)\}$
$\{buy(c_1), buy'(c_1), safe(c_1), exp(c_2), exp(c_3)\}$
$\{exp(c_1), buy(c_2), buy'(c_2), safe(c_2), exp(c_3)\}$
$\{exp(c_1), exp(c_2), buy(c_3), buy'(c_3), safe(c_3)\}$
$\{buy(c_1), buy'(c_1), safe(c_1), buy(c_2), buy'(c_2), safe(c_2), exp(c_3)\}$
$\{exp(c_1), buy(c_2), buy'(c_2), safe(c_2), buy(c_3), buy'(c_3), safe(c_3)\}$
$\{buy(c_1), buy'(c_1), safe(c_1), buy(c_2), buy'(c_2), safe(c_2), buy(c_3), buy'(c_3), safe(c_3)\}$

△

Still before we dwell any deeper in this subject, we define operations useful to project or hide sets of atoms from a module.

Definition 4.3.2 (Hiding and Projecting Atoms) *Let $\mathcal{P} = \langle R, I, O, H \rangle$ be a module and S an arbitrary set of atoms. If we want to* **Hide** *(denoted as $\setminus$) S from program module $\mathcal{P}$, we use*

$$\mathcal{P} \setminus S = \langle R \cup \{\{i\}. \mid i \in I \cap S\}, I \setminus S, O \setminus S, H \cup ((I \cup O) \cap S) \rangle. \tag{4.2}$$

Dually, we can **Project** *(denoted as $|$) over S in the following way:*

$$\mathcal{P} \mid_S = \langle R \cup \{\{i\}. \mid i \in I \setminus S\}, I \cap S, O \cap S, H \cup ((I \cup O) \setminus S) \rangle. \tag{4.3}$$

▲

Both operators *Hide* and *Project* do not change the answer sets of the original program, i.e., $AS(\mathcal{P}) = AS(\mathcal{P} \setminus S) = AS(\mathcal{P}_{|S})$ but do change the set of visible atoms $At_v(\mathcal{P} \setminus S) = At_v(\mathcal{P}) \setminus S$ and $At_v(\mathcal{P} \mid S) = At_v(\mathcal{P}) \cap S$.

Example 4.3.2 (Hiding atoms) *Take again the program module encoding Alice's conditions in Example 2.2.2. Hiding atoms $exp/1$ from it is denoted as $\mathcal{P}_A \setminus \{exp/1\}$ and produces the following module:*

$$\begin{array}{rl} \mathcal{P}_A \setminus \{exp/1\} = < & \{buy(X) \leftarrow car(X), safe(X), not\ exp(X). \\ & \quad car(c_1).\quad car(c_2).\quad car(c_3).\}, \\ & \{safe(c_1), safe(c_2), safe(c_3)\}, \\ & \{buy(c_1), buy(c_2), buy(c_3)\}, \\ & \{car(c_1), car(c_2), car(c_3),\ exp(c_1), exp(c_2), exp(c_3)\} > \end{array}$$

△

4.3.1 Relaxed Output Composition

For the reasons presented before, we start by defining a generalised version of the composition operator, by removing the condition enforcing disjointness of the output signatures of the two modules being combined.

Definition 4.3.3 (Relaxed Composition) *Given two modules $\mathcal{P}_1 = \langle R_1, I_1, O_1, H_1\rangle$ and $\mathcal{P}_2 = \langle R_2, I_2, O_2, H_2\rangle$, their composition $\mathcal{P}_1 \uplus \mathcal{P}_2$ is defined when they respect each others hidden atoms, i.e., $H_1 \cap At(\mathcal{P}_2) = \emptyset$ and $H_2 \cap At(\mathcal{P}_1) = \emptyset$. Then their composition is*

$$\mathcal{P}_1 \uplus \mathcal{P}_2 = \langle R_1 \cup R_2, (I_1 \cup I_2) \setminus (O_1 \cup O_2), O_1 \cup O_2, H_1 \cup H_2\rangle. \tag{4.4}$$

▲

Obviously, the following important properties still hold for $\uplus$:

Lemma 1 *The relaxed composition operator is reflexive, associative, commutative and has the identity element $\langle \emptyset, \emptyset, \emptyset, \emptyset\rangle$.* ○

Proof of Lemma 1. The rules (respectively, the sets of output atoms and the sets of hidden atoms) of the composition are the union of the rules (respectively, the output atoms and the hidden atoms) of the individual modules. The set union operator has the four properties we want to prove and so, they hold for these parts of the composition. As for the input set of the composition, $I_1 \cup I_1 = I_1$, $O_1 \cup O_1 = O_1$ (respectively, $I_1 \cup \emptyset = I_1$, $O_1 \cup \emptyset = O_1$) and the output set is disjoint from the input set hence the input of the composition of I_1 with itself (respectively, the neutral element) remains unchanged. This proves that the operator is: reflexive, associative and commutative and has an identity element. □

Having defined the way to deal with common outputs in the composition of modules, we would like to redefine the operator $\bowtie$ for combining the answer sets of these modules. However, this is shown here to be impossible.

Lemma 2 *The operation $\uplus$ is not compositional, i.e., for any join operation $\bowtie'$, it is not always the case that*

$$AS(\mathcal{P}_1 \uplus \mathcal{P}_2) = AS(\mathcal{P}_1) \bowtie' AS(\mathcal{P}_2). \tag{4.5}$$

○

Proof of Lemma 2. A join operation is a function mapping a pair of sets of interpretations into a set of interpretations. Consider the following program modules:

$\mathcal{P}_1 = < \{a.\}, \emptyset, \{a,b\}, \emptyset >$	$\mathcal{Q}_1 = < \{a.\ \ \bot \leftarrow a, b.\}, \emptyset, \{a,b\}, \emptyset >$
$\mathcal{P}_2 = \langle \{b.\}, \emptyset, \{b\}, \emptyset\rangle$	$\mathcal{Q}_2 = \langle \{b.\}, \emptyset, \{b\}, \emptyset\rangle$
$\mathcal{P}_1 \uplus \mathcal{P}_2 = < \{a.\ \ b.\}, \emptyset, \{a,b\}, \emptyset >$	$\mathcal{Q}_1 \uplus \mathcal{Q}_2 = < \{a.\ \ \bot \leftarrow a, b.\quad b.\}, \emptyset, \{a,b\}, \emptyset >$

One sees that $AS(\mathcal{P}_1) = AS(\mathcal{Q}_1) = \{\{a\}\}$, and $AS(\mathcal{P}_2) = AS(\mathcal{Q}_2) = \{\{b\}\}$ but $AS(\mathcal{P}_1 \uplus \mathcal{P}_2) = \{\{a,b\}\}$ while $AS(\mathcal{Q}_1 \uplus \mathcal{Q}_2) = \{\}$. Therefore, it cannot exist $\bowtie'$ since this would require $AS(\mathcal{P}_1 \uplus \mathcal{P}_2) = AS(\mathcal{P}_1) \bowtie' AS(\mathcal{P}_2) = \{\{a\}\} \bowtie' \{\{b\}\} = AS(\mathcal{Q}_1) \bowtie' AS(\mathcal{Q}_2) = AS(\mathcal{Q}_1 \uplus \mathcal{Q}_2)$, a contradiction. □

As we have motivated in the introduction, it is important to applications to be able to use $\uplus$ to combine program modules, and retain some form of compositionality. The following definition presents a construction that adds the required information in order to be able to combine program modules using the original natural join.

Definition 4.3.4 (Transformed Relaxed Composition) *Consider the program modules $\mathcal{P}_1 = \langle R_1, I_1, O_1, H_1 \rangle$ and $\mathcal{P}_2 = \langle R_2, I_2, O_2, H_2 \rangle$. Let $O = O_1 \cap O_2$, and define the sets of newly introduced atoms $O' = \{o' \mid o \in O\}$ and $O'' = \{o'' \mid o \in O\}$. Construct program module:*

$$\begin{array}{l} \mathcal{P}_{union} = < R_{union}, O' \cup O'', O, \emptyset > \text{ where:} \\ R_{union} = \{o \leftarrow o'. \mid o' \in O'\} \cup \{o \leftarrow o''. \mid o'' \in O''\}. \end{array}$$

The transformed relaxed composition is defined as the program module

$$(\mathcal{P}_1 \uplus^{RT} \mathcal{P}_2) = [\rho_{O' \leftarrow O}(\mathcal{P}_1) \sqcup \rho_{O'' \leftarrow O}(\mathcal{P}_2) \sqcup \mathcal{P}_{union}] \setminus [O' \cup O''] \tag{4.6}$$

▲

Intuitively, we rename the common output atoms in the original modules, and introduce an extra program module that unites the contributions of each module by a pair of rules for each common atom $o \leftarrow o'$ and $o \leftarrow o''$. We then hide all the auxiliary atoms to obtain the original visible signature. If $O = \emptyset$ then $\mathcal{P}_{union}$ is empty, and all the other modules are not altered, falling back to the original definition.

Example 4.3.3 (Transformed Relaxed Composition) *Recall modules $\mathcal{P}_B$ and $\mathcal{P}_C$ in Example 4.2.1:*

$$\mathcal{P}_A = \left\langle \begin{array}{l} R = \{buy(X) \leftarrow car(X), safe(X), not\ exp(X). \\ \qquad car(c_1). \quad car(c_2). \quad car(c_3).\}, \\ I = \{\ safe(c_1), safe(c_2), safe(c_3), exp(c_1), exp(c_2), exp(c_3)\}, \\ O = \{buy(c_1), buy(c_2), buy(c_3)\}, \\ H = \{car(c_1), car(c_2), car(c_3)\} \end{array} \right\rangle$$

$$\mathcal{P}_B = \left\langle \begin{array}{l} R = \{exp(c_2).\}, \\ I = \{\ \}, \\ O = \{exp(c_2), exp(c_3)\}, \\ H = \{\} \end{array} \right\rangle$$

$$\mathcal{P}_C = \left\langle \begin{array}{ll} R = \{ & exp(c_3).\}, \\ I = \{ & \}, \\ O = \{ & exp(c_1), exp(c_2), exp(c_3)\}, \\ H = \{ & \} \end{array} \right\rangle$$

From these, we construct the following program modules:

$$\rho_{o'\leftarrow o}(\mathcal{P}_A) = \left\langle \begin{array}{ll} R = \{ & buy'(X) \leftarrow car(X), safe(X),\ not\ exp(X). \\ & car(c_1).\quad car(c_2).\quad car(c_3). \\ & \bot \leftarrow buy(X)', not\ buy(X).\}, \\ I = \{ & buy(X), safe(c_1), safe(c_2), safe(c_3), \\ & exp(c_1), exp(c_2), exp(c_3)\}, \\ O = \{ & buy'(c_1), buy'(c_2), buy'(c_3)\}, \\ H = \{ & car(c_1), car(c_2), car(c_3)\} \end{array} \right\rangle$$

$$\rho_{o''\leftarrow o}(\mathcal{P}_B) = \left\langle \begin{array}{ll} R = \{ & exp(c_2)''\}, \\ I = \{ & exp(c_2)\}, \\ O = \{ & exp(c_2)'', exp(c_3)''\}, \\ H = \{ & \} \end{array} \right\rangle$$

$$\mathcal{P}_{union} = \left\langle \begin{array}{ll} R = \{ & buy(c_1) \leftarrow buy'(c_1). \\ & buy(c_2) \leftarrow buy'(c_2). \\ & buy(c_3) \leftarrow buy'(c_3). \\ & exp(c_2) \leftarrow exp''(c_2).\ exp(c_3) \leftarrow exp''(c_3).\}, \\ I = \{ & buy'(c_1), buy'(c_2), buy'(c_3), exp''(c_2), exp''(c_3)\}, \\ O = \{ & exp(c_2),\ exp(c_3)\}, \\ H = & \emptyset \end{array} \right\rangle$$

And their transformed relaxed composition is then:

$$\mathcal{P}_A \uplus^{RT} \mathcal{P}_B = \left\langle \begin{array}{ll} R = \{ & buy'(X) \leftarrow car(X), safe(X),\ not\ exp(X). \\ & car(c_1).\quad car(c_2).\quad car(c_3). \\ & \bot \leftarrow buy(X)', not\ buy(X). \\ & exp''(c_2). \\ & buy(c_1) \leftarrow buy'(c_1). \\ & buy(c_2) \leftarrow buy'(c_2).\ buy(c_3) \leftarrow buy'(c_3). \\ & exp(c_2) \leftarrow exp''(c_2).\ exp(c_3) \leftarrow exp''(c_3).\}, \\ I = \{ & buy(X), safe(c_1), safe(c_2), safe(c_3), \\ & exp(c_1), exp(c_2), exp(c_3)\}, \\ O = \{ & buy'(c_1), buy'(c_2), buy'(c_3), exp''(c_2), exp''(c_3)\}, \\ H = \{ & car(c_1), car(c_2), car(c_3)\} \end{array} \right\rangle$$

△

Theorem 4.3.1 *Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be arbitrary program modules without positive dependencies among them. Then, modules joined with operators $\uplus$ and $\uplus^{RT}$ are modularly equivalent:*

$$\mathcal{P}_1 \uplus \mathcal{P}_2 \equiv_m \mathcal{P}_1 \uplus^{RT} \mathcal{P}_2. \tag{4.7}$$

○

Proof of Theorem 4.3.1. By reduction of the conditions of the theorem to the conditions necessary for applying the original Module Theorem. If $\mathcal{P}_1 \uplus \mathcal{P}_2$ is defined then let their transformed relaxed composition be $T = (\mathcal{P}_1 \uplus^{RT} \mathcal{P}_2)$. It is clear that the output atoms of T are $O_1 \cup O_2$, the input atoms are $(I_1 \cup I_2) \setminus (O_1 \cup O_2)$, and the hidden atoms are $H_1 \cup H_2 \cup O' \cup O''$. Note that before the application of the hiding operator the output atoms are $O_1 \cup O_2 \cup O' \cup O''$. The original composition operator $\sqcup$ can be applied since the outputs of $\rho_{O' \leftarrow O}(\mathcal{P}_1)$, $\rho_{O'' \leftarrow O}(\mathcal{P}_2)$ and $\mathcal{P}_{union}$ are respectively $O' \cup (O_1 \setminus O)$, $O'' \cup (O_2 \setminus O)$ and $O = O_1 \cap O_2$, which are pairwise disjoint.

Because of this, we are in the conditions of the original Module Theorem and thus it is applicable to the result of the modified composition $\uplus$ iff the transformation did not introduce positive loops between the program parts of the three auxiliary models. If $\mathcal{P}_1 \uplus \mathcal{P}_2$ had no loops between the common output atoms than its transformation $\mathcal{P}_1 \uplus^{RT} \mathcal{P}_2$ also does not because it results from a renaming into new atoms.

Consider now the rules part of T; if we ignore the extra introduced atoms in O' and O'' the program obtained has exactly the same answer sets of the union of program parts of $\mathcal{P}_1$ and $\mathcal{P}_2$. Basically, we are substituting the union of $o \leftarrow Body_1^1., \ldots, o \leftarrow Body_m^1.$ in $\mathcal{P}_1$, and $o \leftarrow Body_1^2., \ldots, o \leftarrow Body_n^2.$ in $\mathcal{P}_2$ by:

$o \leftarrow o'.$	$o \leftarrow o''.$
$o' \leftarrow Body_1^1.$	$o'' \leftarrow Body_1^2.$
$\ldots$	$\ldots$
$o' \leftarrow Body_m^1.$	$o'' \leftarrow Body_n^2.$
$\bot \leftarrow o', not\ o.$	$\bot \leftarrow o'', not\ o.$

This guarantees visible equivalence of $\mathcal{P}_1 \uplus \mathcal{P}_2$ and $\mathcal{P}_1 \uplus^{RT} \mathcal{P}_2$, since the models of each combined modules are in one-to-one correspondence, and they coincide in the visible atoms. The contribution of the common output atoms is recovered by the joins involving atoms in O', O'' and O, that are all pairwise disjoint, and ensuring that answer sets obey to $o = o' \vee o''$ via program module $\mathcal{P}_{union}$. The constraints introduced in the transformed models $\rho_{O' \leftarrow O}(\mathcal{P}_1)$ (respectively, $\rho_{O'' \leftarrow O}(\mathcal{P}_2)$) simply prune models that have o false and o' (respectively, o'') true, reducing the number of models necessary to consider. Since the input and output atoms of $\mathcal{P}_1 \uplus \mathcal{P}_2$ and $\mathcal{P}_1 \uplus^{RT} \mathcal{P}_2$ are the same, then $\mathcal{P}_1 \uplus \mathcal{P}_2 \equiv_m \mathcal{P}_1 \uplus^{RT} \mathcal{P}_2$. □

The important remark is that according to the original module theorem we have:

$$AS(\rho_{O' \leftarrow O}(\mathcal{P}_1) \sqcup \rho_{O'' \leftarrow O}(\mathcal{P}_2) \sqcup \mathcal{P}_{union}) =$$

$$AS(\rho_{O' \leftarrow O}(\mathcal{P}_1)) \bowtie AS(\rho_{O'' \leftarrow O}(\mathcal{P}_2)) \bowtie AS(\mathcal{P}_{union})$$

Therefore, from a semantical point of view, users can always substitute module $\mathcal{P}_1 \uplus \mathcal{P}_2$ by $\mathcal{P}_1 \uplus^{RT} \mathcal{P}_2$, which has an extra cost since the models of the renamed program modules may increase. This is, however, essential to regain compositionality.

Example 4.3.4 *Considering the following program modules:*

$$\mathcal{Q}_1 = < \{a. \quad \bot \leftarrow a, b.\}, \emptyset, \{a, b\}, \emptyset >$$

and:

$$\mathcal{Q}_2 = \langle \{b.\}, \emptyset, \{b\}, \emptyset \rangle$$

We have that:

$$\rho_{a',b' \leftarrow a,b}(\mathcal{P}_1) = \left\langle \begin{array}{ll} \{ & a'.\ \bot \leftarrow a', not\ a. \\ & \bot \leftarrow b', not\ b.\}, \\ \{ & a,b\}, \{a',b'\}, \emptyset \end{array} \right\rangle$$

$$\rho_{a'',b'' \leftarrow a,b}(\mathcal{P}_2) = \left\langle \begin{array}{ll} \{ & b''.\ \bot \leftarrow a'', not\ a. \\ & \bot \leftarrow b'', not\ b.\}, \\ \{ & a,b\}, \{a'',b''\}, \emptyset \end{array} \right\rangle$$

$$\mathcal{P}_{union} = \left\langle \begin{array}{ll} \{ & a \leftarrow a'.\ a \leftarrow a''. \\ & b \leftarrow b'.\ b \leftarrow b''.\}, \\ \{ & a',a'',b',b''\}, \{a,b\}, \emptyset \end{array} \right\rangle$$

$$\rho_{a',b' \leftarrow a,b}(\mathcal{Q}_1) = \left\langle \begin{array}{ll} \{ & a'.\ \bot \leftarrow a,b. \\ & \bot \leftarrow a', not\ a. \\ & \bot \leftarrow b', not\ b.\}, \\ \{ & a,b\}, \{a',b'\}, \emptyset \end{array} \right\rangle$$

$$\begin{array}{ll} \rho_{a'',b'' \leftarrow a,b}(\mathcal{Q}_2) = & \rho_{a'',b'' \leftarrow a,b}(\mathcal{P}_2) \\ \mathcal{Q}_3 = \mathcal{P}_{union} & \end{array}$$

The answer sets of the first two modules are $\{\{a,a'\},\{a,b,a'\}\}$ *and* $\{\{b,b''\},\{a,b,b''\}\}$*, respectively. Their join is* $\{\{a,b,a',b''\}\}$ *and the returned model belongs to* $\mathcal{P}_{union}$ *(and thus it is compatible), and corresponds to the only intended model* $\{a,b\}$ *of* $\mathcal{P}_1 \uplus \mathcal{P}_2$*. Note that the answer sets of* $\mathcal{P}_{union}$ *are 16, corresponding to the models of propositional formula* $(a \equiv a' \vee a'') \wedge (b \equiv b' \vee b'')$*. Regarding, the transformed module* $\rho_{a',b' \leftarrow a,b}(\mathcal{Q}_1)$ *it discards the model* $\{a,b,a'\}$*, having answer sets* $\{\{a,a'\}\}$*. But now the join is empty, as intended.* ■

4.3.2 Conservative Output Composition

In order to preserve the original outer join operator, which is widely used in databases, for the form of composition we introduce next, one must redefine the original composition operator ($\oplus$). We do that by resorting to a program transformation such that the composition operator remains compositional with respect to the join operator ($\bowtie$). The transformation we present next consists of taking Definition 4.3.4 and adding an extra module to guarantee that only compatible models (models that coincide on the visible part) are retained.

Definition 4.3.5 (Conservative Composition) *Let* $\mathcal{P}_1 = \langle R_1, I_1, O_1, H_1 \rangle$ *and* $\mathcal{P}_2 = \langle R_2, I_2, O_2, H_2 \rangle$ *be modules such that* $O = O_1 \cap O_2 \neq \emptyset$*. Let* $O' = \{o' \mid o \in O\}$ *and* $O'' = \{o'' \mid o \in O\}$ *be sets of newly introduced atoms.*

Construct program modules:

$$\mathcal{P}_{union} = \langle R_{union}, O' \cup O'', O, \emptyset \rangle$$
Where R_{union} is a set of rules.
$$R_{union} = \{o \leftarrow o'. \mid o' \in O'\} \cup \{o \leftarrow o''. \mid o'' \in O''\}$$
and:
$$\mathcal{P}_{filter} = \langle \{\bot \leftarrow o', not\ o''.\ \bot \leftarrow not\ o', o''. \mid o \in O\}, O' \cup O'', \emptyset, \emptyset \rangle$$

The conservative composition is defined as the following program module:

$$\mathcal{P}_1 \otimes \mathcal{P}_2 = [(\rho_{O' \leftarrow O}(\mathcal{P}_1) \sqcup \rho_{O'' \leftarrow O}(\mathcal{P}_2) \sqcup \mathcal{P}_{union} \sqcup \mathcal{P}_{filter}] \setminus (O' \cup O'') \tag{4.8}$$

▲

Note that when the intersection of the output sets is empty, the definition corresponds to the case when we are combining a module with the neutral element. Note further here that each rule not containing atoms that belong to $O_1 \cap O_2$ in $\mathcal{P}_1 \cup \mathcal{P}_2$ is included in $\mathcal{P}_1 \otimes \mathcal{P}_2$. Therefore, it is easy to see that this transformational semantics ($\otimes$) is a conservative extension to the existing one ($\oplus$).

Theorem 4.3.2 (Conservative Module Theorem) *If $\mathcal{P}_1, \mathcal{P}_2$ are modules such that $\mathcal{P}_1 \otimes \mathcal{P}_2$ is defined, then a model $M \in AS(\mathcal{P}_1 \otimes \mathcal{P}_2)$ iff $M \cap (At(\mathcal{P}_1) \cup At(\mathcal{P}_2)) \in AS(\mathcal{P}_1) \bowtie AS(\mathcal{P}_2)$.* ∘

Proof of Theorem 4.3.2. The theorem states that if we ignore the renamed literals in $\otimes$ the models are exactly the same, as expected. The transformed program module $\mathcal{P}_1 \otimes \mathcal{P}_2$ corresponds basically to the union of programs, as seen before. Consider a common output atom o. The constraints in the module part $\mathcal{P}_{filter}$ combined with the rules in $\mathcal{P}_{union}$ restrict the models to the cases for which $o \equiv o' \equiv o''$. The equivalence $o \equiv o'$ restricts the answer sets of $\rho_{o' \leftarrow o}(\mathcal{P}_1)$ to the original answer sets (except for the extra atom o') of $\mathcal{P}_1$, and similarly the equivalence $o \equiv o''$ filters the answer sets of $\rho_{o'' \leftarrow o}(\mathcal{P}_2)$ obtaining the original answer sets of $\mathcal{P}_2$. Now it is immediate to see that compositionality is retained by making the original common atoms o compatible. □

The above theorem is very similar to the original Module Theorem of Oikarinen and Janhunen apart from the extra renamed atoms required in $\mathcal{P}_1 \otimes \mathcal{P}_2$ to obtain compositionality.

Shortcomings Revisited This Section 4.3 provides a means to deal with the restriction that we identified and that disallows the composition of modules with common outputs. Take the following example:

Example 4.3.5 *Returning to the introductory example, we can conclude that $\mathcal{P}_{mg_1} \otimes \mathcal{P}_{mg_2}$ has only one answer set:*

$$\{safe(c_1), airbag(c_1), car(c_1), car(c_2), car(c_3)\}$$

since this is the only compatible model between $\mathcal{P}_{mg_1}$ and $\mathcal{P}_{mg_2}$. The answer sets of $\rho(\mathcal{P}_{mg_1})$ and $\rho(\mathcal{P}_{mg_2})$, are collected in the table below where compatible models appear in the same row and $car(c_1), car(c_2), car(c_3)$ has been omitted from $AS(\rho(\mathcal{P}_{mg_2}))$. Predicate s (respectively, a) stands for safe (respectively, airbag).

Answer sets of $\rho(\mathcal{P}_{mg_1})$	*Answer sets of* $\rho(\mathcal{P}_{mg_2})$
$\{s(c_1), s'(c_1)\}$	$\{s(c_1), s''(c_1), a(c_1)\}$
$\{s(c_1), s(c_2), s'(c_1)\}$	$\{s(c_1), s(c_2), s''(c_1), a(c_1)\}$
$\{s(c_1), s(c_3), s'(c_1)\}$	$\{s(c_1), s(c_3), s''(c_1), a(c_1)\}$ $\{s(c_1), s(c_3), s''(c_1),$ $s''(c_3), a(c_1), a(c_3)\}$
$\{s(c_1), s(c_2), s(c_3),$ $s'(c_1)\}$	$\{s(c_1), s(c_2), s(c_3),$ $s''(c_1), a(c_1)\}$ $\{s(c_1), s(c_2), s(c_3), s''(c_1),$ $s''(c_3), a(c_1), a(c_3), c(c_1)\}$

The only compatible model retained after composing with $\mathcal{P}_{union}$ and $\mathcal{P}_{filter}$ is the combination of the answer sets in the first row:

$$\{s(c_1), s'(c_1), s''(c_1), a(c_1), c(c_1), c(c_2), c(c_3)\}.$$

Naturaly, this corresponds to the intended result if we ignore the s' and s'' atoms. ■

We underline that models of composition $\mathcal{P}_1 \otimes \mathcal{P}_2$ will either contain all atoms o, o', and o'' or none of them, and will only join compatible models from $\mathcal{P}_1$ having $\{o, o'\}$ with models in $\mathcal{P}_2$ having $\{o, o''\}$, or models without atoms in $\{o, o', o''\}$.

4.3.3 Complexity

Regarding complexity, checking the existence of $M \in P_1 \oplus P_2$ and $M \in P_1 \uplus^{RT} P_2$ is an NP-complete problem. It is immediate to define a decision algorithm belonging to Σ_2^p that checks existence of an answer set of the module composition operators. This is strictly less than the results in the approach of Dao-Tran *et al.* (2009) where the existence decision problem for propositional theories is NEXP^{NP}-complete – however their approach allows disjunctive rules.

4.4 Compositional Semantics for Modular Logic Programming

As we have seen, the existing compositional semantics does not work directly for modules with common outputs. However, it becomes clear from Section 4.3 that the effect of the extra renamed modules $\mathcal{P}_{union}$ and $\mathcal{P}_{filter}$ is to impose extra conditions on the compatible models of the other two modules, which are the same for both composition forms. In fact, by changing the definition of answer sets of a module we can obtain a fully compositional semantics.

Definition 4.4.1 (Modular models of a generalised MLP) *Given a program module $\mathcal{P} = \langle R, I, O, H \rangle$ its modular models are $MM(\mathcal{P}) = AS(\rho_{O^s \leftarrow O}(\mathcal{P}))$.* ▲

The following Lemmas relates the modular models with the answer sets of a program module, and thus provides a way of recovering the answer sets from modular models.

Lemma 3 *Let $\mathcal{P}$ be a program module $\mathcal{P} = \langle R, I, O, H \rangle$. Then, $M \in AS(\mathcal{P})$ iff $(M \cup M^s) \in MM(\mathcal{P})$ where $M^s = \{o^s \mid o \in O \text{ and } o \in M\}$.* ∘

Proof of Lemma 3. Replacing the definitions: $M \in LM(R_I^M \cup \{a. \mid a \in M \cap I\})$ iff $(M \cup \{o^s \mid o \in O \text{ and } o \in M\}) \in LM(R_I^{s,M} \cup \{\bot \leftarrow o^s, not\ o.\} \cup \{a. \mid a \in M \cap I\}$. We can easily see that by adding renamed output atoms to the model on the left hand side we will only allow supported models to obey the constraint on the right hand side and vice-versa. All the remaining is the same. □

Lemma 4 *Consider program module $\mathcal{P} = \langle R, I, O, \emptyset \rangle$. Then, $M \in MM(\mathcal{P})$ iff $M \cap (I \cup O)$ is a classical model of R. Moreover, atom o is supported in M iff $o^s \in M$.* ∘

Proof of Lemma 4. The first part of this lemma comes directly from the definition that $M \in AS(\rho_{O^s \leftarrow O}(\mathcal{P})) \Leftrightarrow LM\left(\rho(R)^M_{I \cup O} \cup \{a. | a \in M \cap (I \cup O)\}\right)$. As for the second part, this is a necessity since otherwise M would not be a model due to $\bot \leftarrow o^s, not\ o$ and vice-versa. □

Our Modular Models are models of the original program with extra annotations conveying if an atom is supported or not. This is a little more complex when hidden atoms are involved, but the interpretation of the extra atoms is the same: whenever o^s is true then there is a rule for o with true body, otherwise o^s is false. We now define the join operators for the two forms of composition presented in Section 4.3:

Definition 4.4.2 *Given two modules $\mathcal{P}_1 = \langle R_1, I_1, O_1, H_1 \rangle$ and $\mathcal{P}_2 = \langle R_2, I_2, O_2, H_2 \rangle$ and sets of MMs $A_1 \subseteq 2^{I_1 \cup O_1 \cup O_1^s}$ and $A_2 \subseteq 2^{I_2 \cup O_2 \cup O_2^s}$. Let:*

$$
\begin{aligned}
A_1 \bowtie^+ A_2 = \{M_1 \cup M_2 \mid\ & M_1 \in A_1, M_2 \in A_2, \text{ and} \\
& M_1 \cap (I_2 \cup O_2) = M_2 \cap (I_1 \cup O_1)\} \\
A_1 \bowtie^\times A_2 = \{M_1 \cup M_2 \mid\ & M_1 \in A_1, M_2 \in A_2, \\
& M_1 \cap (I_2 \cup O_2 \cup O_2^s) = M_2 \cap (I_1 \cup O_1 \cup O_1^s)\}
\end{aligned}
$$

▲

So, when joining two models we either look at visible atoms of the original modules for the case of the relaxed composition, or look at all visible atoms and extra annotations, thus discarding non-supported modular models. The main result is as follows:

Theorem 4.4.1 *If $\mathcal{P}_1, \mathcal{P}_2$ are modules such that $\mathcal{P}_1 \uplus \mathcal{P}_2$ and $\mathcal{P}_1 \otimes \mathcal{P}_2$ is defined, then:*

$$(1)\ MM(\mathcal{P}_1 \uplus \mathcal{P}_2) = MM(\mathcal{P}_1) \bowtie^{+} MM(\mathcal{P}_2)$$
$$(2)\ MM(\mathcal{P}_1 \otimes \mathcal{P}_2) = MM(\mathcal{P}_1) \bowtie^{\times} MM(\mathcal{P}_2)$$

∘

Proof of Theorem 4.4.1.

(1) Is by Definition 4.4.1 equal to $AS(\rho_{O^s \leftarrow O}(\mathcal{P}_1 \uplus \mathcal{P}_2)) = AS(\rho_{O^s \leftarrow O}(\mathcal{P}_1)) \bowtie^{+} AS(\rho_{O^s \leftarrow O}(\mathcal{P}_2))$. Let now M, M_1 and M_2 respectively belong to $AS(\rho_{O^s \leftarrow O}(\mathcal{P}_1 \uplus \mathcal{P}_2)$, $AS(\rho_{O^s \leftarrow O}(\mathcal{P}_1))$ and $AS(\rho_{O^s \leftarrow O}(\mathcal{P}_2))$.

($\rightarrow$) Now, we have that M_1 (respectively, M_2) is $M \cap At_v(\rho(P_1)) = M \cap (I_1 \cup O_1 \cup H_1 \cup O^s)$ (respectively, $M \cap At_v(\rho(P_2)) = M \cap (I_2 \cup O_2 \cup H_2 \cup O^s)$). Models M_1 and M_2 are compatible by Definition 4.4.2 if $M \cap M_1 \cap (I_2 \cup O_2) = M \cap M_2 \cap (I_1 \cup O_1)$. Take the ones that are compatible. M_1 will now be an AS of P_1 if it is a model and it is minimal. It is easy to see that it is a model since the outputs of P_1 and P_2 were made disjoint and M is supported by their composition and so, the rules supporting M_1 must come from P_1. As for the minimality it is the case that for every $o^s \in O^s$, either: **(i)** $o^s \in M$ implying that $o \in M$. For atom o to occur in M, it either belongs to M_1, to M_2 or to both. There are no occurrences of o^s in the bodies of rules (by construction) so if there are rules r that are satisfied in P_1 s.t. $Head(r) = o^s$ then o^s (and thus o) $\in M_1$ and they have support in P_1. Atom o can also have no support in rules but vary by belonging to the input of $\mathcal{P}_1$ and/or P_2. Or: **(ii)** $o \notin M$ implies that o^s is not in M_1 nor M_2.

It follows, because M is minimal, that M_1 and M_2 are MM of their respective modular reducts, thus answer sets of $\mathcal{P}_1$ and P_2 and that their join $\bowtie^{+}$ is M.

($\leftarrow$) For M_1 and M_2 to be compatible according to $\bowtie^{+}$, $M_1 \cap (I_2 \cup O_2) = M_2 \cap (I_1 \cup O_1)$ meaning that all inputs and outputs must have the same truth value. Now, $M = M_1 \cup M_2$ is an AS of the composition if it is a model of the composition (atoms in M_1 and in M_2 are supported) and it is a minimal model. M_1 is a model of R_1 and M_2 a model of R_2 and because all bodies of these rules coincide with M_1 and M_2 (because they are compatible) and all ICs ($\bot \leftarrow o, not\ o^s$ and $\bot \leftarrow not\ o, o^s$) are satisfied by construction, M is a model of $\rho_{O^s \leftarrow O}(\mathcal{P}_1 \uplus \mathcal{P}_2)$. Model M would not be minimal only if there was an o^s in it that (loosely speaking) did not need to be. That would mean that all rules $r : o^s \leftarrow body$ would have body false but that is not possible since that would make either M_1 or M_2 not minimal. Thus M is also a minimal model and an AS.

(2): having proven (1), it suffices to see that $\bowtie^{\times}$ makes us look at annotations O^s for support, which discards all non supported models. That also happens on the left hand side of the equality since $\otimes$ imposes the same restrictions with the introduction of ICs in extra module 4. Since the models were the same to begin with (before looking at annotations), and we discard all models that are not supported in terms of o and o^s, we end up with the same models on both sides of the equality be-

cause $AS(\rho_{O^s \leftarrow O}(\mathcal{P}_1))$ will not find support in $AS(\rho_{O^s \leftarrow O}(\mathcal{P}_2))$ since the renaming is different (and vice-versa). □

Example 4.4.1 (Modular Models) *Take again Example 4.3.4. Recall the following program module:*

$$\mathcal{Q}_1 = \langle \{a. \quad \bot \leftarrow a, b.\}, \emptyset, \{a, b\}, \emptyset \rangle$$

Its renaming transformation is:

$$\rho_{a',b' \leftarrow a,b}(\mathcal{Q}_1) = \left\langle \begin{array}{ll} \{ & a'.\ \bot \leftarrow a, b. \\ & \bot \leftarrow a', not\ a. \\ & \bot \leftarrow b', not\ b.\}, \\ \{ & a, b\}, \{a', b'\}, \emptyset \end{array} \right\rangle$$

The transformed module $\rho_{a',b' \leftarrow a,b}(\mathcal{Q}_1)$ has a single answer set $\{\{a, a'\}\}$ which corresponds to MModel $M \in MM(\mathcal{Q}_1) = \{\{a, a^s\}\}$ ■

4.5 Conclusions and Future Work

We redefined the necessary operators in order to relax the conditions for combining modules with common atoms in their output signatures. Two alternative solutions are presented, both allowing us to retain compositionality while dealing with a more general setting than before. Dao-Tran *et al.* (2009) provide an embedding of the original composition operator of Oikarinen and Janhunen into their approach. Since our constructions rely on a transformational approach using operator $\sqcup$ of Oikarinen and Janhunen, by composing both translations, an embedding into Dao-Tran *et al.* (2009) is immediately obtained. Tasharrofi & Ternovska (2011) take Janhunen *et al.* (2009) and extend it with an algebra which includes a new operation of feedback (loop) over modules. They have shown that the loop operation adds significant expressive power – modules can can express all (and only) problems in NP. The other issues remain unsolved though.

The module theorem has been extended to the general theory of answer sets Babb & Lee (2012), being applied to non-ground logic programs containing choice rules, the count aggregate, and nested expressions. It is based on the new findings about the relationship between the module theorem and the splitting theorem Lifschitz & Turner (1994). It retains the composition condition of disjoint outputs and still forbids positive dependencies between modules. As for disjunctive versions, Janhunen *et al.* (2009) introduced a formal framework for modular programming in the context of DLPs under stable model semantics. This is based on the notion of DLP-functions, which resort to appropriate input/output interfacing. Similar module concepts have already been studied for the cases of normal logic programs and answer set programs and even propositional theories, but the special characteristics of disjunctive rules are properly taken into account in the syntactic and semantic definitions of DLP functions

presented therein. In Gebser *et al.* (2011a), MLP is used as a basis for Reactive Answer Set Programming, aiming at reasoning about real-time dynamic systems running online in changing environments.

As future work we can straightforwardly extend these results to probabilistic reasoning with answer sets by applying the new module theorem to Damásio & Moura (2011) (see Chapter 6), as well as to DLP functions and general answer sets. An implementation of the framework is also planned in order to assess the overhead when compared with the original benchmarks in Oikarinen & Janhunen (2008).

Based on the work we present in Section 4.4, together with results in the literature, we believe that a fully compositional semantics can be attained by resorting to partial interpretations e.g., SE-models Turner (2003b) for defining program models at the semantic level. It is known that one must include extra information about the support of each atom in the models in order to attain generalised compositionality and SE-models are known to be sufficient while it may be the case that they are more expressive than actually needed.

Chapter 5

Allowing Positive Cyclic Dependencies Between Modules

In this Chapter 5 we discuss two alternative solutions to the cyclic dependencies problem, generalising the module theorem by allowing positive loops between atoms in the interface signatures of the modules being composed.

The work we discuss in this chapter relies on the concepts introduced previously in Sections 2.2 and 4.2, where we presented an overview of the modular logic programming paradigm, identifying its two shortcomings. We begin with an introduction in Section 5.1 where we promptly present a relaxation of the module theorem that will be necessary in the following Section 5.2, where we discuss alternative methods for lifting the restriction that disallows positive cyclic dependencies. We finish the chapter with conclusions and a general discussion.

5.1 Introduction

To attain a generalised form of compositionality we need to be able to deal with both restrictions identified previously and in particular cyclic dependencies between modules. In the literature, Dao-Tran *et al.* (2009) present a solution based on a model minimality property. It forces one to check for minimality on every comparable models of all program modules being composed. It is not directly applicable to our setting though, which can be seen here in Example 5.1.1 where logical constant $\bot$ represents value $false$.

Example 5.1.1 *Given MLP program modules* $\mathcal{P}_1 = \langle\{a \leftarrow b.\ \bot \leftarrow not\ b.\}, \{b\}, \{a\}, \{\}\rangle$, *having one answer set* $\{a,b\}$, *and* $\mathcal{P}_2 = \langle\{b \leftarrow a.\}, \{a\}, \{b\}, \{\}\rangle$ *having two answer sets, namely* $\{\}$ *and* $\{a,b\}$, *their MLP composition has no inputs and no intended answer sets while their (minimal) join is* $\{a,b\}$, *according to Dao-Tran* et al. *(2009).* ■

Oikarinen and Janhunen's Modular Logic Programming approach is limited by module conditions that are imposed in order to ensure the compatibility of their

module system with the answer set semantics, namely forcing output signatures of composing modules to be disjoint and disallowing positive cyclic dependencies between different modules. These conditions are too restrictive in practice and after discussing alternative ways of lifting the first restriction in the previous Chapter 4, we now show in this current Chapter 5 how one can allow positive cyclic dependencies between modules, thus widening the applicability of this framework and the scope of the module theorem. We can furthermore assume that every pair of modules under composition have disjoint outputs, either because they were disjoint to begin with or they were made disjoint after applying the results we presented in the previous chapter.

Graph Definitions We define next what a dependency graph is as well as what supported and self-supported (or unsupported) positive cyclic dependencies are.

Definition 5.1.1 (Dependency Graph) *The positive dependency graph of a finite program P, denoted by $Dep^+(P)$, is a finite directed graph with signed edges. Its nodes are the relations occurring in P. For every clause in P which uses relation p in its head and relation q in a positive (resp. negative) literal in its body, there is a positive (resp. negative) edge (p, q) in $Dep^+(P)$. We say then that p uses q positively (resp. negatively).*

– We say that p depends positively (resp. negatively) on q if there is a path in $Dep^+(P)$, from p to q with only positive edges (resp. at least one negative edge).

– We say that p depends evenly (resp. oddly) on q if there is a path in $Dep^+(P)$, from p to q with an even (resp. odd) number of negative edges. ▲

We recall now that a ***strongly connected component*** (SCC) of a directed graph is a maximal subset of mutually reachable nodes and use this to define next the concepts of self-supported and well-supported positive loops. The graph formed by these SCCs (each SCC as an individual node) and their dependencies (as arcs) is a directed acyclic graph (DAG) to which we refer by **Condensed Graph**.

Definition 5.1.2 (Supported Positive Cyclic Dependencies) *Given a logic program P and an interpretation I, a* well-supported*, or simply a supported positive loop (or positive cyclic dependency) is formed when an atom belonging to a strongly connected component SCC_1 of the dependency graph $Dep^+(P)$ has a rule $r \in P$ which is applicable with respect to interpretation I such that $a \in Head(r)$ and every atom $b \in Body(r)$ is such that $b \notin SCC_1$.* ▲

Definition 5.1.3 (Self-supported Positive Cyclic Dependencies) *Given a logic program P and an interpretation I, a* self-supported *positive loop (or just, positive cyclic dependency) is formed when no atom belonging to a strongly connected component SCC_1 of the dependency graph $Dep^+(P)$ has a rule $r \in P$ which is applicable with respect to interpretation I such that $a \in Head(r)$ and every atom $b_i \in Body(r)$ is such that $b_i \notin SCC_1$.* ▲

The compatibility criterion for the join operator $\bowtie$ rules out the compositionality of mutually dependent modules, but allows positive loops inside modules or negative loops in general. We recall the issue with self-supported positive loops between modules in Example 2.2.7 for which problem we present a solution in this Chapter 5.

Example 5.1.2 (Self-Supported Cyclic Dependencies) *Take the following two program modules stating that "all safe cars have airbags" and that "a car is safe if it has an airbag":*

$$\mathcal{P}_1 = \langle \{airbag \leftarrow safe.\}, \{safe\}, \{airbag\}, \emptyset \rangle$$
$$\mathcal{P}_2 = \langle \{safe \leftarrow airbag.\}, \{airbag\}, \{safe\}, \emptyset \rangle$$

Their answer sets are:

$$AS(\mathcal{P}_1) = AS(\mathcal{P}_2) = \{\{\}, \{airbag, safe\}\}$$

while the single answer set of the composition $\mathcal{P}_1 \oplus \mathcal{P}_2$ (there are no remaining input atoms to vary) is the empty model $\{\}$. Therefore:

$$AS(\mathcal{P}_1 \sqcup \mathcal{P}_2) \neq AS(\mathcal{P}_1) \bowtie AS(\mathcal{P}_2) = \{\{\}, \{airbag, safe\}\}$$

Thus, because the inputs of the composed module give support to the loop, this also invalidates the module theorem.

Note that the program union contains a single SCC containing airbag and safe which effectively forms a self-supported positive loop with respect to interpretation $\{airbag, safe\}$ but no self-supported loops with respect to to interpretation $\{\}$ which corresponds to its single answer set, while $\mathcal{P}_1$ and $\mathcal{P}_2$ contain each two SCCs, both of them containing single atoms (namely airbag in one and safe in the other), thus having no loops with respect to interpretations $\{\}$ and $\{airbag, safe\}$ which correspond to their answer sets. ■

This example sheds light over the real problem, which lies not in positive loops in general but rather, more specifically, in positive self-supported loops which are introduced when mutually dependent modules are composed. If we were to exclude this type of loops, the original MLP framework could potentially cope with well supported loops if one were to be able to identify and isolate the said self supported loops.

Consider now the following modified version of the previous example, depicting a supported positive loop (which we also call well supported positive loop):

Example 5.1.3 (Well Supported Cyclic Dependencies) *Take the following two program modules, the first of which stating that "cars that are safe have airbags" and that "new cars have airbags" while the second states that "a car is safe if it has an airbag" and that "a car is safe if it has* Electronic Stability Program (ESP)*":*

$$\mathcal{P}_1 = \langle \{airbag \leftarrow safe.\ \ airbag \leftarrow new.\}, \{safe, new\}, \{airbag\}, \emptyset \rangle$$
$$\mathcal{P}_2 = \langle \{safe \leftarrow airbag.\ \ safe \leftarrow esp.\}, \{airbag, esp\}, \{safe\}, \emptyset \rangle$$

Their answer sets are:

$$\begin{aligned} AS(\mathcal{P}_1) = &\quad \{\{\}, \{airbag, safe\}, \{airbag, new\}, \{airbag, safe, new\}\} \\ AS(\mathcal{P}_2) = &\quad \{\{\}, \{airbag, safe\}, \{esp, safe\}, \{airbag, esp, safe\}\} \end{aligned}$$

while the answer sets of the composition of programs $AS(\mathcal{P}_1 \oplus \mathcal{P}_2)$ *(making inputs atoms* new *and* esp *vary) are:*

$$AS(\mathcal{P}_1 \sqcup \mathcal{P}_2) = \{\{\}, \{airbag, safe, esp\}, \{airbag, new, safe, esp\}, \{airbag, new, safe\}\}$$

Therefore, the following would hold if well supported positive loops between modules were to be allowed:

$$AS(\mathcal{P}_1 \oplus \mathcal{P}_2) = AS(\mathcal{P}_1) \bowtie AS(\mathcal{P}_2).$$

Note now that the program union contains now three SCCs, one containing airbag and $safe$ *and the other two containing respectively new and esp. This effectively forms a supported positive loop with respect to interpretations* $\{\}$, $\{airbag, safe, esp\}$, $\{airbag, new, safe, esp\}$ *and* $\{airbag, new, safe\}$ *which all correspond to its answer sets. The individual modules* $\mathcal{P}_1$ *and* $\mathcal{P}_2$ *contain each also three SCCs, all of them containing single atoms (namely airbag,* $safe$ *and new in one and airbag,* $safe$ *and esp in the other), thus having no loops with respect to the intended interpretations* $\{\}, \{airbag, safe, esp\}, \{airbag, new, safe, esp\}$ *and* $\{airbag, new, safe\}$ *which all correspond to their respective answer sets.*

■

We capture this in the next theorem, relaxing the original applicability conditions of the module theorem.

Theorem 5.1.1 (Relaxed Module Theorem) *The module theorem (Theorem 2.2.1) holds in the presence of supported positive loops between modules if we allow a new operator* $\sqcup_r$ *to compose them by removing item (ii) from its conditions in Definition 2.2.4.* ○

Proof of Theorem 5.1.1. Let modules $\mathcal{P}_1$ and $\mathcal{P}_2$ be such that they have no self-supported positive cyclic dependencies, but may have supported positive cyclic dependencies, between their inputs and outputs. Let us denote their relaxed composition by $\mathcal{P}_C = \mathcal{P}_1 \sqcup_r \mathcal{P}_2$.

$\rightarrow$ Towards a contradiction lets assume the following:

$$AS(\mathcal{P}_C) \neq AS(\mathcal{P}_1) \bowtie AS(\mathcal{P}_2)$$

There is an answer set M of $\mathcal{P}_C$ which is not in $AS(\mathcal{P}_1) \bowtie AS(\mathcal{P}_2)$. This means that, $M \in LM(\langle R_1 \cup R_2, (I_1 \setminus O_2) \cup (I_2 \setminus O_1), O_1 \cup O_2, H_1 \cup H_2 \rangle)$, $M = M_1 \cup M_2$ where $M_1 = M \cap At(\mathcal{P}_1)$ and $M_2 = M \cap At(\mathcal{P}_2)$ and one of the following:

- $M_1 \notin AS(\mathcal{P}_1)$ (respectively, $M_2 \notin AS(\mathcal{P}_2)$), which would mean that either $M_1 \not\models \mathcal{P}_1$ (respectively, $M_2 \not\models \mathcal{P}_2$) or M_1 (respectively, M_2) is not minimal;

  Let us start by considering the choice rules that are introduced for input atoms. The union of the choice rules in $\mathcal{P}_1$ with the ones in $\mathcal{P}_2$ is potentially different from the set of choice rules in the union $\mathcal{P}_C$ (due to some of the inputs of one of the modules being *met* by the outputs of the other and vice versa). If we consider the truth values of each atom in model M and make the corresponding choice rules in modules $\mathcal{P}_1$ and $\mathcal{P}_2$ take the corresponding values in order to coincide with M, then it is the case that M_1 and M_2 are respectively models of $\mathcal{P}_1$ and $\mathcal{P}_2$.

  Having fixed the values for the choice rules, every other rule r in P_i, for $i \in \{1,2\}$, is also in $\mathcal{P}_C$, thus if r is applicable with respect to M than it is also applicable with respect to M_i because M_i is a model over a vocabulary restricted to the vocabulary of P_i.

  As for minimality, choice rules that are not in $\mathcal{P}_C$ (the ones that were *met* by the union of modules) are always, by definition, relative to output atoms from $\mathcal{P}_2$ (respectively, $\mathcal{P}_1$) that appear in the head of some rule $r_{out} \in \mathcal{P}_2$ (respectively, $r_{out} \in \mathcal{P}_1$). If we consider that the value of these atoms is fixed in M and that they appear in the body of some rule $r_{in} \in \mathcal{P}_1$ (respectively, $r_{in} \in \mathcal{P}_2$), this effectively isolates $\mathcal{P}_1$ from $\mathcal{P}_2$ in the union of modules $\mathcal{P}_C$ with respect to model M. As such if some M_i is not minimal, then so is not M.

- $M_1 \cap At(\mathcal{P}_2) \neq M_2 \cap At(\mathcal{P}_1)$, which would mean that the modules are actually not compatible which contradicts our assumption.

Because of this, equality in first direction always holds.

$\leftarrow$ We proceed with a proof by induction over the structure of the condensed graph of $\mathcal{P}_C$, which we call $CDep^+(\mathcal{P}_C)$ which, we recall, is a directly acyclic graph (DAG):

For the **Base Cases** consider strongly connected components SCC_i (of the dependency graph $CDep^+(\mathcal{P}_C)$) not depending on any other SCC_j:

1. SCC_i contains a single atom which can be either:

   (i) a hidden atom a_{hid}, in which case if there is a fact for it in the reduct $(\mathcal{P}_C)^M$, then the same fact occurs in either the reduct of $\mathcal{P}_1$ with respect to M_1 ($\mathcal{P}_1^{M_1}$) or in the reduct of $\mathcal{P}_2$ with respect to M_2 ($\mathcal{P}_2^{M_2}$), and the converse holds when there is not a fact for a_{hid} in the reduct $(\mathcal{P}_C)^M$, then such fact also does not occur in either $\mathcal{P}_1^{M_1}$ or in $\mathcal{P}_2^{M_2}$. If the atom is not hidden, it must be:

   (ii) an input atom which we call a_{in}. There are two sub cases:

- the first in which there is an un-met choice rule in $\mathcal{P}_C$ which, for M to be in the join of M_1 and M_2, takes the same value in the union of the modules and in its corresponding original module;

- and the second, in which there is an output atom of one of the modules (call it $\mathcal{P}_1$ without loss of generality) $a_{in} \in At_{Out}(\mathcal{P}_1)$). We know that there is no rule in $\mathcal{P}_1$ which makes a_{in} true with respect to the M_1 (and thus there is also no rule in $\mathcal{P}_C$ which makes a_{in} true with respect to the M) because otherwise this would reflect in the dependency graph.

2. SCC_i contains more than an atom, all of which belong to the hidden signature of one of the original modules (say P_i). This SCC is necessarily self-supported which reflects in every atom in this SCC being false in M and in both M_1 and M_2. This is the case of a self-supported positive loop inside one of the original modules.

3. SCC_i contains more than an atom and some of them belong to the signatures of both modules $\mathcal{P}_1$ and $\mathcal{P}_2$. This is the case of a self-supported mutual dependency between the two modules and is discarded by the hypotheses of the theorem.

Inductive Hypotheses: Given a strongly connected component SCC_i, every other SCC_j from which it depends[1] (either directly or indirectly) take the same values, atom wise, with respect to model M and with respect to both models M_1 and M_2.

For the **Inductive Steps** consider strongly connected components SCC_i (of the condensed graph $CDep^+(\mathcal{P}_C)$) having atoms that depend on atoms outside their own SCC_i. Note that there are also three cases for the inductive step, which correspond to the three base cases (but where atoms in SCC_i have external dependencies).

1. SCC_i contains a single atom a for which either:
   (i) there is an applicable rule r in reduct $\mathcal{P}_1^{M_1}$ (respectively, in $\mathcal{P}_2^{M_2}$) that supports a ($a \in Head(r)$ and the body is satisfied by the model). Then it must be the case that $r \in (\mathcal{P}_C)^M$. Thus, by construction, every atom b in $Body(r)$ is such that b belongs to an SCC_j below SCC_i. Then, by induction hypotheses, a is true in M_1 which implies that it is also true in M.
   (ii) the converse holds when there is no applicable rule r in reduct $\mathcal{P}_1^{M_1}$ (respectively, $\mathcal{P}_2^{M_2}$) and, hence, a has no support.

2. SCC_i contains multiple atoms, all of which belong to the hidden signature of the same module: $a_1 \in At_h(P_i), \ldots, a_n \in At_h(P_i)$, $i \in \{1,2\}$. For each of these atoms, say $a_j \in M_i$ such that $j \in \{1,\ldots,n\}$:
   (i) if a is true in M_i then there in a rule r in $\mathcal{P}_i^{M_i}$ which is applicable with respect to M_i and supports a_j. Therefore, it is also the case that $r \in (\mathcal{P}_C)^M$.

[1] We say that SCC_j is *below* SCC_i

Thus, similarly to step 1.(i), by construction every atom b in $Body(r)$ is such that if b belongs to a different strongly connected component SCC_j, then SCC_j must be below SCC_i to which a_j belongs. Then, by induction hypotheses, this atom a_j is true in M_1 which implies that it is true in M.

(ii) If on the contrary a_j is false in the model, then the converse holds.

3. SCC_i contains more than an atom and some of them belong to the signatures of both modules $\mathcal{P}_1$ and $\mathcal{P}_2$. This case is the important case that deals with supported positive mutual dependencies between the modules under composition. Consider the following atoms in SCC_i:

$$a_1 \in At(\mathcal{P}_1), \ldots, a_n \in At(\mathcal{P}_1),$$

$$b_1 \in At(\mathcal{P}_1), b_1 \in At(\mathcal{P}_2), \ldots, b_m \in At(\mathcal{P}_1), b_m \in At(\mathcal{P}_2),$$

$$c_1 \in At(\mathcal{P}_2), \ldots, c_o \in At(\mathcal{P}_2)$$

For each of these atoms, say a_i such that $i \in \{1, \ldots, n\}$ (respectively, c_k such that $k \in \{1, \ldots, o\}$ or b_j such that $j \in \{1, \ldots, m\}$), if a_i (respectively, c_k or b_j) is true then there in an applicable rule r in $\mathcal{P}_1^{M_1}$ (respectively, r in $\mathcal{P}_2^{M_2}$, or in the case of b_j r in $\mathcal{P}_i^{M_i}$ where $b_j \in At_{Out}(\mathcal{P}_i)$, and $i \in \{1,2\}$) with respect to its corresponding model, which gives support to this atom. Therefore, it is also the case that $r \in (\mathcal{P}_C)^M$. Thus, similarly to step 2.(i) and 2.(ii), by construction every atom d in $Body(r)$ is such that if d belongs to a different strongly connected component SCC_j, then SCC_j must be below SCC_i to which a_i (respectively, c_k) belongs. Then, by induction hypotheses, these atoms a_i are true in M_1 (respectively, c_k is true in M_2 or b_j is true in both M_1 and M_2) which implies that they are true in M.

Note that we discarded self-supported loops which implies that one of these atoms indeed is supported by an applicable rule with respect to to one of the models M_1 or M_2 and thus also M.

(ii) If on the contrary a_i (respectively, c_k or b_j) is false in the model, then the converse holds.

Because of this, equality in the second direction always holds as well. □

5.2 Positive Cyclic Dependencies Between Modules

5.2.1 Model Minimisation for Join Operator

We present a model join operation for definite programs that requires one to look at every model of both modules being composed in order to check for minimality on models comparable on account of their input signatures. This operation is able to distinguish between atoms that are self supported through positive loops and atoms

with proper support, allowing one to lift the condition in Definition 2.2.4 disallowing positive dependencies between modules for some cases.

Definition 5.2.1 (Join) *Given modules $\mathcal{P}_1$ and $\mathcal{P}_2$, let their composition be $\mathcal{P}_C = \mathcal{P}_1 \oplus \mathcal{P}_2$. We define their minimal join as:*

$$AS(\mathcal{P}_1) \bowtie^{min} AS(\mathcal{P}_2) = \{M \mid M \in AS(\mathcal{P}_1) \bowtie AS(\mathcal{P}_2) \text{ suchthat}$$

$$\nexists_{M' \in AS(\mathcal{P}_1) \bowtie AS(\mathcal{P}_2)} : M' \subset M \text{ and } M \cap At_{in}(\mathcal{P}_C) = M' \cap At_{in}(\mathcal{P}_C)\}$$

▲

Example 5.2.1 (Minimal Join) *A car is safe if it has an airbag and it has an airbag if it is safe and the airbag is an available option. This is captured by two modules, namely:*

$$\mathcal{P}_1 = \langle \{airbag \leftarrow safe, available\_option.\}, \{safe, available\_option\}, \{airbag\}, \emptyset \rangle$$

$$\mathcal{P}_2 = \langle \{safe \leftarrow airbag.\}, \{airbag\}, \{safe\}, \emptyset \rangle$$

Which respectively have:

$$AS(\mathcal{P}_1) = \{\{\},\ \{safe\},\ \{available\_option\},\ \{airbag, safe, available\_option\}\}$$

$$AS(\mathcal{P}_2) = \{\{\}, \{airbag, safe\}\}$$

The composition has as its input signature $\{available\_option\}$ and therefore its answer set $\{airbag,safe,available\_option\}$ is not minimal regarding the input signature of the composition because $\{available\_option\}$ is also an AS (and the only intended model among these two). Thus,

$$AS(\mathcal{P}_1 \oplus \mathcal{P}_2) = AS(\mathcal{P}_1) \bowtie^{min} AS(\mathcal{P}_2) = \{\{\}, \{available\_option\}\}$$

■

This join operator allows us to lift the prohibition of composing mutually dependent modules under certain situations. Integrity constraints containing only input atoms in their body are still a problem with this approach as these exclude models that would otherwise be minimal in the presence of self-supported loops. Because of this, we need to discuss a more complex solution in the next section, that requires introducing further information in the models.

Example 5.2.2 (Problem with Negative Rules) *Consider the following two program modules, one of which containing an integrity constraint:*

$$P_1 = \left\langle \begin{array}{ll} R = & \{a \leftarrow b.\ \ \leftarrow not\ b.\}, \\ I = & \{b\}, \\ O = & \{a\} \\ H = & \{\} \end{array} \right\rangle$$

having one answer set, i.e., $AS(P_1) = \{\{a,b\}\}$*, and*

$$P_2 = \left\langle \begin{array}{ll} R = & \{b \leftarrow a.\}, \\ I = & \{a\} \\ O = & \{b\} \\ H = & \{\} \end{array} \right\rangle$$

having two answer sets, i.e., $AS(P_2) = \{\{\},\{a,b\}\}$

Their composition is:

$$P_C = \left\langle \begin{array}{ll} R = & \{a \leftarrow b.\ \ b \leftarrow a.\ \ \leftarrow not\ b.\} \\ I = & \emptyset \\ O = & \{a,b\} \\ H = & \{\} \end{array} \right\rangle$$

This composition P_C *has no intended answer sets while the minimal join of the models of the composing modules is* $\{a,b\}$ △

The following theorem shows that, for positive programs, a minimisation is sufficient to lift the applicability of MLP to mutually dependent modules.

Theorem 5.2.1 (Minimal Module Theorem for Positive Programs) *If* $\mathcal{P}_1, \mathcal{P}_2$ *are modules such that* $\mathcal{P}_1 \oplus \mathcal{P}_2$ *is defined (allowing cyclic dependencies between modules), and that only positive rules are used in modules, then:*

$$AS(\mathcal{P}_1 \oplus \mathcal{P}_2) = AS(\mathcal{P}_1) \bowtie^{min} AS(\mathcal{P}_2)$$

○

Proof of Theorem 5.2.1. Let us start by assuming there are no cyclic dependencies between the modules. Then, because we know from the original module theorem that the answer sets of the composition are the same as the join of the individual sets of answer sets:

$$AS(\mathcal{P}_1 \oplus \mathcal{P}_2) = AS(\mathcal{P}_1) \bowtie AS(\mathcal{P}_2)$$

and that given an input set, by definition of an AS they are all minimal relative to set inclusion, the minimisation introduced by the minimal join will not eliminate any model.

Let us now assume there are cyclic dependencies between the modules. Towards a contradiction assume that this is a self-supported loop (as per Definition 5.1.3) with respect to one model of the composition (belonging to the join of models), such that it is not removed by the minimality condition. This means that, because it is an answer set, this model would be minimal relative to its corresponding set of inputs of the composition. For it not to be an intended model due to being unsupported then some atom in the model is supported by a set of atoms in the input of one of the modules that are not inputs of the composition (because the rules of the composition are the rules

of the original programs plus choice rules for the inputs). There will thus be another model of the same original module where all these input atoms are false. Because of the join conditions, this model (joined with the corresponding one from the other module) is minimal regarding the inputs which contradicts our assumption. □

5.2.2 Annotated Models For Dealing With Positive Loops

Because the former operator is not general and it forces us to compare each model with every other model for minimality, thus not being "local", and furthermore it does not fully solve the problem, we present next an alternative that requires adding annotations to models.

We need a way to identify self-supported positive cyclic dependencies (loops) that are formed by composition by looking only at one model from each module under composition. It is known from the literature (e.g., Slota & Leite (2012)) that in order to do without looking at the rules of the program modules being composed, which in the setting of MLP we have to assume not even having access to, we need to have extra information incorporated into their models.

We start by defining next what is an annotated interpretation, which maps every atom into a set of subsets of input atoms tracking the dependencies of the atom from combinations of its input atoms, following by a fixed point definition of an annotated model given an annotated reduct. After that we define a form of compatibility for annotated models and with that criterion, define a loop tolerant join operator which allows us to redefine the module theorem to a completely general setting.

Definition 5.2.2 (Annotated Interpretation) *Given a program module $\mathcal{P}$, we define annotated interpretations I^A of $\mathcal{P}$ as functions mapping sets of atoms to sets of subsets of input atoms as follows:*

$$I^A : At(\mathcal{P}) \rightarrow 2^{2^{At_{in}(\mathcal{P})}}$$

such that the mapping for an atom $a \in At_{in}(\mathcal{P})$ is either $I^A(a) = \{\}$ or $I^A(a) = \{\{a\}\}$ when a is set as false (respectively, true) in the input signature of $\mathcal{P}$. The set of all interpretations is a complete lattice with the following partial order relation:

$$I^A \preceq J^A \text{ iff } \forall a : a \in At(\mathcal{P}) \implies I^A(a) \subseteq J^A(a)$$

▲

We denote an atom a annotated with a set of subsets of input atoms D as a_D. We introduce next an operator that projects annotated atoms from annotated interpretations.

Definition 5.2.3 (Projection operator $Int(I^A)$) *Given an annotated interpretation I^A, we obtain a standard interpretation from it by discarding its annotations as follows:*

$$Int(I^A) = \{a \mid I^A(a) \neq \{\}\}$$

▲

We define next an annotated model as the least model of a fixed point operator iterating over the rules of a module.

Definition 5.2.4 (Annotated Model) *Given a definite program P, a set of (input) atoms In and an annotated interpretation I^A as defined before, the annotated model over annotations of P will be the least fixed point of the immediate consequences monotonic operator defined as follows for every atom $a \in At(\mathcal{P})$:*

$$T^A_{P,In}(I^A)(a) = \bigcup \begin{cases} \{D_1 \cup \ldots \cup D_m \mid a \leftarrow L_1, \ldots, L_m \in \mathcal{P} \wedge \\ \qquad D_1 \in I^A(L_1) \wedge \ldots \wedge D_m \in I^A(L_m)\} & , \; if \; a \notin In, \; and \\ \{\{a\}\} & , \; (if \; a \in In) \; otherwise. \end{cases}$$

The annotated least model (ALM) of program $\mathcal{P}$ is thus:

$$ALM_{P,In} = lfp(T^A_{P,In}) = T^A_{P,In} \uparrow^{\lambda}$$

for some ordinal λ s.t. we start from $T^A_{P,In} \uparrow^0$ where for every atom a,

$$T^A_{P,In} \uparrow^0 (a) = \{\}$$

And update it with:

$$\begin{array}{ll} T^A_{P,In} \uparrow^{\alpha+1} = T^A_{P,In}(T^A_{P,In} \uparrow^{\alpha}) & \text{if } \alpha+1 \text{ is a successor ordinal, and} \\ T^A_{P,In} \uparrow^{\alpha} = \bigcup_{\beta<\alpha} T^A_{P,In} \uparrow^{\beta} & \text{if } \alpha \text{ is a limit ordinal.} \end{array}$$

▲

As we only consider positive programs, and because the annotations are also finite (albeit exponential), the operator reaches a fixed point after a finite number of iterations. Naturally, the operator is monotonic, which means that given two annotated interpretations, if one precedes the other in terms of annotation ordering, then so do their annotated fixed points. We capture this in the following theorem.

Lemma 5 (Monotony of T^A_P)

$$If \; I^A \preceq^A J^A \; then \; T^A_P(I^A) \preceq^A T^A_P(J^A)$$

∘

Proof of Lemma 5. Having:

$$I^A \preceq J^A$$

implies that it is the case that for every atom $L_i \in At$:

$$I^A(L_i) \subseteq J^A(L_i)$$

Which means that, because the program $\mathcal{P}$ is positive, given rules like the following:

$$a \leftarrow L_1, \ldots, L_m \in \mathcal{P}$$

for which:

$$D_1 \in I^A(L_1) \wedge \ldots \wedge D_m \in I^A(L_m)$$

it is then the case that:

$$D_1 \cup \ldots \cup D_m \in I^A$$

Because of the initial assumption, this further implies that

$$D_1 \cup \ldots \cup D_m \in J^A$$

Taking the previous Definition 5.2.4, we can conclude that:

$$\forall a \notin In, \text{ if } D_1 \cup \ldots \cup D_m \in T_P^A(I^A)(a) \text{ then } D_1 \cup \ldots \cup D_m \in T_P^A(J^A)(a)$$

and

$$\forall a \in In, \text{ if } \{a\} \in T_P^A(I^A)(a) \text{ then } \{a\} \in T_P^A(J^A)(a)$$

Because the dependencies for individual atoms in the body of the rule are in the interpretations, we have that the following is then generically the case:

$$T_P^A(I^A)(a) \subseteq T_P^A(J^A)(a)$$

Which means that the annotated interpretation for a given atom a in a given annotated interpretation I^A one obtains by iterating the T_P^A operator until reaching the fixed point is always contained in the interpretation obtained by iterating the operator given a superset interpretation J^A.

Then:

$$T_P^A(I^A) \preceq^A T_P^A(J^A)$$

□

We now state an important lemma establishing the correspondence between annotated models and classical models. It will later become clear that this property is crucial for our goal of allowing positive loops between modules.

Lemma 6 (Correspondence of $T_{P,In}^A$ and $T_{P \cup In}$) *Given a definite program P, and a set of (true input) atoms In, the projection of each annotated fixed point corresponds to a fixed point of the original operator $T_{P \cup In}$ for every atom $a \in At(P)$:*

$$a \in Int(lfp(T_{P,In}^A)) \text{ iff } a \in lfp(T_{P \cup In})$$

○

Proof of Lemma 6. We prove the lemma by fixed point induction.

Base Case: For every a, $a \notin Int(T^A_{P,In}\uparrow^0)$ and $a \notin T_{P\cup In}\uparrow^0$.

Inductive Hypothesis: Assume that $a \in Int(T^A_{P,In}\uparrow^n)$ iff $a \in T_{P\cup In}\uparrow^n$.

Inductive Case: We will prove that $\forall a \in At(P)$, $a \in Int(T^A_{P,In}\uparrow^{n+1})$ iff $a \in T_{P\cup In}\uparrow^{n+1}$.

Case 1 (Suppose $a \in In$): There is a fact a. for it in $P \cup In$. Then, by definition $T^A_{P,In}\uparrow^{n+1}(a) = \{\{a\}\}$ hence it is the case that $a \in Int(T^A_{P,In}\uparrow^{n+1})$. Because it is also the case that $a \in T_{P\cup In}\uparrow^{n+1}(a)$, the steps match.

Case 2 (Suppose $a \notin In$): We know that:

(i) By definition of the T^A_P operator:

$$a \in Int(T^A_{P,In}\uparrow^{n+1})$$

if there is a dependency:

$$D_1 \cup \ldots \cup D_m$$

in the set of dependencies for a such that rule:

$$r : a \leftarrow L_1, \ldots, L_m. \in P$$

and

$$D_1 \in T^A_{P,In}\uparrow^n(L_1) \wedge \ldots \wedge D_m \in T^A_{P,In}\uparrow^n(L_m).$$

(ii) Now, by definition of the T_P operator,

$$a \in T_{P\cup In}\uparrow^{n+1}$$

if it is the case that a is derived from at least one rule satisfied in P such that:

$$a \leftarrow L_1, \ldots, L_m.$$

and:

$$L_1 \in T_{P\cup In}\uparrow^n \wedge \ldots \wedge L_n \in T_{P\cup In}\uparrow^n .$$

We proceed to prove the two directions separately as follows:

(If $a \in T_{P\cup In}\uparrow^{n+1} \Longrightarrow a \in Int(T^A_{P,In}\uparrow^{n+1})$): Assuming:

$$a \in T_{P\cup In}\uparrow^{n+1}$$

Implies that there is a rule r, satisfied in P:

$$r : a \leftarrow L_1, \ldots, L_m. \in P$$

Such that:

$$L_1 \in T_{P\cup In}\uparrow^n \wedge \ldots \wedge L_m \in T_{P\cup In}\uparrow^n .$$

Now, by the inductive hypothesis:

$$D_1 \in T^A_{P,In} \uparrow^n (L_1) \wedge \ldots \wedge D_m \in T^A_{P,In} \uparrow^n (L_m).$$

hence, the dependency set for a:

$$D_1 \cup \ldots \cup D_m$$

appears in the set of dependencies ($\bigcup$ in the definition of the operator) for a, meaning that:

$$a \in Int(T^A_{P,In} \uparrow^{n+1}).$$

(If $a \in Int(T^A_{P,In} \uparrow^{n+1}) \Longrightarrow a \in T_{P \cup In} \uparrow^{n+1}$): Assuming:

$$a \in Int(T^A_{P,In} \uparrow^{n+1}).$$

Implies that there is a satisfied rule r:

$$r : a \leftarrow L_1, \ldots, L_m. \in P$$

Such that:

$$D_1 \in T^A_{P,In} \uparrow^n (L_1) \wedge \ldots \wedge D_m \in T^A_{P,In} \uparrow^n (L_m).$$

Now, by hypothesis:

$$L_1 \in T_{P \cup In} \uparrow^n (L_1) \wedge \ldots \wedge L_m \in T_{P \cup In} \uparrow^n .$$

Which means that it is also the case that that every atom $at \in Body(r)$ is satisfied by $T_{P \cup In} \uparrow^n$: hence a, is $satisfied$ and as such:

$$a \in T_{P \cup In} \uparrow^{n+1} .$$

Hence $a \in Int(T^A_{P,In} \uparrow^{n+1})$ iff $a \in T_{P \cup In} \uparrow^{n+1}$.

Note that it might be the case that $T^A_{P,In}$ stops after more steps than $T_{P \cup In}$, intuitively because the alternative annotations for alternative rules must propagate after step n when the projected model $Int(T^A_{P,In} \uparrow^n)$ is already fixed. It is the case though that because we are facing positive programs with finite (albeit possibly exponential) annotations, the annotated operator will always converge for computations when the original operator also converges. □

The next Lemma states that a model is annotated correctly if all of its atoms are annotated only with subsets of a conjunction of its inputs.

Lemma 7 (Correctness of $T^A_{P,In}$) *Let P be a definite program, and In be a set of (input) atoms. Given any $C \subseteq In$ then it is the case that any atom $a \in lfp(T_{P \cup C})$ iff $\exists D_j \in lfp(T^A_{P,In})(a)$ and $D_j \subseteq C$.* ∘

Proof of Lemma 7. We prove the lemma by fixed point induction.

Base Case: For every atom a, $\nexists D_j \in T^A_{P,In}\uparrow^0(a)$ and $a \notin T_{P\cup C}\uparrow^0$ such that $D_j \subseteq C$.

Inductive Hypothesis: Assume that $a \in T_{P\cup C}\uparrow^n$ iff $\exists D_j \in T^A_{P,In}\uparrow^n(a)$ such that $D_j \subseteq C$.

Inductive Case: We will prove that $\exists D_j \in T^A_{P,In}\uparrow^{n+1}(a)$ iff $a \in T_{P\cup C}\uparrow^{n+1}$ such that $D_j \subseteq C$.

Case 1 (Suppose $a \in C$): There is a fact $a.$ for it in $P \cup C$. Then, by definition

$$T^A_{P,In}\uparrow^{n+1}(a) = \{\{a\}\}$$

hence it is the case that

$$D_j \in T^A_{P,In}\uparrow^{n+1}(a),\ D_j = \{a\} \text{ and } D_j \subseteq C$$

Because it is also true that

$$a \in T_{P\cup C}\uparrow^{n+1},$$

this case holds.

Case 2 (Suppose $a \notin C$): We proceed to prove the two directions separately as follows:

(If $a \in T_{P\cup C}\uparrow^{n+1} \Longrightarrow \exists D_j \in T^A_{P,In}\uparrow^{n+1}$): We begin by assuming:

$$a \in T_{P\cup C}\uparrow^{n+1}$$

which implies that there is a rule r, satisfied in $P \cup C$:

$$r : a \leftarrow L_1, \ldots, L_m. \in P \cup C$$

such that the following holds:

$$L_1 \in T_{P\cup C}\uparrow^n \wedge \ldots \wedge L_m \in T_{P\cup C}\uparrow^n.$$

Now, by induction hypothesis, it is also the case that:

$$\exists D_1 \in T^A_{P,In}\uparrow^n(L_1) \wedge \ldots \wedge \exists D_m \in T^A_{P,In}\uparrow^n(L_m), s.t.\ D_1 \subseteq C, \ldots, D_m \subseteq C.$$

hence, it is the case that there is a dependency set $D_j = D_1 \cup \ldots \cup D_m$ for a that is contained in C and because of that:

$$\exists D_j \in T^A_{P,In}\uparrow^{n+1} \text{ and } D_j \subseteq C$$

which proves this direction.

(If $\exists D_j \in T^A_{P,In}\uparrow^{n+1} \Longrightarrow a \in T_{P\cup C}\uparrow^{n+1}$): We begin by assuming:

$$\exists D_j \in T^A_{P,In}\uparrow^{n+1} \text{ and } D_j \subseteq C$$

which implies that there is a satisfied rule r:

$$r : a \leftarrow L_1, \ldots, L_m. \in P$$

such that the following holds:

$$\exists D_1 \in T^A_{P,In}\uparrow^n (L_1) \wedge \ldots \wedge \exists D_m \in T^A_{P,In}\uparrow^n (L_m), s.t.\ D_1 \subseteq C, \ldots, D_m \subseteq C.$$

Now, by hypothesis:

$$L_1 \in T_{P\cup C}\uparrow^n \wedge \ldots \wedge L_m \in T_{P\cup C}\uparrow^n .$$

Which means that it is also the case that that every atom $at \in Body(r)$ is satisfied by $T_{P\cup C}\uparrow^n$: hence a, is *satisfied* and as such:

$$a \in T_{P\cup C}\uparrow^{n+1}$$

Hence $a \in Int(T^A_{P,In}\uparrow^{n+1})$ iff $a \in T_{P\cup C}\uparrow^{n+1}$ and $D_j \subseteq C$. Which proves this direction, and because both directions hold, proves also this case. □

We introduce now the notion of reconstructed program which, intuitively, corresponds to a positive program having one rule for each of the dependencies of an atom in a given annotated model. If an atom has no dependencies but belongs to the model, then a fact is added.

Definition 5.2.5 (Reconstructed program) *Given a module $\mathcal{P}$, a program $P_{rec}(M)$ can conversely be reconstructed from one of the module's annotated models M simply as a set of rules $r_1, \ldots, r_m$ for each annotated atom $a_{\{D_1,\ldots,D_m\}} \in M$ s.t. a is not an input atom ($a \notin At_{in}(\mathcal{P})$), of the following form:*

$$\begin{array}{c} Head(r_1) = a \text{ and } Body(r_1) = D_1. \\ \vdots \\ Head(r_m) = a \text{ and } Body(r_m) = D_m. \end{array} \tag{5.1}$$

▲

Note that, because atoms in the bodies of these rules are all input atoms (and thus, there are no rules defining them in the reconstructed program), reconstructed programs are loop free.

Such **reconstructed program** $P_{rec}(M)$ will be equivalent (but not strongly equivalent in the sense of Lifschitz *et al.* (2000)) to taking the original program and adding facts that belong to the annotated model M, intersected with the input signature of $\mathcal{P}$,

correspondingly. We present an example of a reconstructed program after defining what are annotated answer sets.

We define next a reduct operator, able to deal with annotated interpretations, in a way similar to the original Gelfond-Lifschitz reduct which then allows us to define annotated answer sets.

Definition 5.2.6 (Annotated Reduct) *Given a program module $\mathcal{P}$ and an annotated interpretation I^A, we define $\mathcal{P}^{I^A}$ as follows:*

$$\mathcal{P}^{I^A} = \{Head(r) \leftarrow Body^+(r) \mid r \in R(\mathcal{P}), Body^-(r) \cap Int(I^A) = \emptyset\} \tag{5.2}$$

▲

Definition 5.2.7 (Annotated Answer Sets) *Take a program module P and its answer sets $AS(P)$. We define its annotated answer sets as :*

$$AS^A(P) = \{M_i^A \mid M_i \in AS(P) \ \wedge \ M_i^A \ = ALM_{\mathcal{P}^{I^A},\ M \cap At_{in}(\mathcal{P})}\}$$

▲

Example 5.2.3 (Annotated Answer Sets and Reconstructed Programs) *Consider the following program module:*

$$\mathcal{P} = \langle \{a \leftarrow b, c. \quad b \leftarrow d, not\ e, not\ f.\}, \{d, f\},\ \{a, b\},\ \{c, e\} \rangle$$

$\mathcal{P}$ has four annotated models, corresponding to the four alternative input combinations, as per Definition 5.2.7:

$$AS^A(\mathcal{P}) = \{\{b_{\{\{d\}\}}, d_{\{\{d\}\}}\},\ \{f_{\{\{f\}\}}\},\ \{d_{\{\{d\}\}}, f_{\{\{f\}\}}\}\},\ \{\}\}$$

Now take the annotated model $M = \{b_{\{\{d\}\}}, d_{\{\{d\}\}}\}$. A program $P_{rec}(M)$ reconstructed from model M, as per Definition 5.2.5, is as follows: $P_{rec}(M) = \{b \leftarrow d.\}$.

△

In the previous Example 5.2.3, the first rule $a \leftarrow b, c.$ can never be satisfied because c is not an input atom ($c \notin I$) and it is not satisfied by the rules of the module ($R_{\mathcal{P}} \not\models c$). Thus, the only potential positive loop is identified by $\{b_{\{\{d\}\}}, d_{\{\{d\}\}}\}\}$. If we compose $\mathcal{P}$ with e.g., module $\mathcal{P}_{loop} = \langle \{d \leftarrow b.\},\ \{b\},\ \{d\},\ \emptyset \rangle$, having one annotated model $\{d_{\{\{b\}\}}, b_{\{\{b\}\}}\}$, then it becomes possible to identify that this composition produces a positive loop and because of this only if some atom in the loop is satisfied by the module composition, there will be an answer set reflecting that.

Also note that since e is not a visible atom ($e \in H$), it does not interfere with other modules, as long as it is respected, and thus it does not need to be in the annotation. As for f, it does not need to be in the annotation of answer sets containing b because it appears negated and the join operator for models will disallow its join with models where it appears as being true.

5.2.2.1 Cyclic Compatibility

Next we define the compatibility of mutually dependent models. We assume that the outputs are disjoint as per the original MLP definitions, which can always be achieved with the work we presented before in Chapter 4. The compatibility is defined as a two step criterion. The first is similar to the original compatibility criterion, only adapted to dealing with annotated models by disregarding the annotations. This first step makes annotations of negative dependencies unnecessary. The second step disallows the join of models that contain atoms belonging to a self-supported positive loop.

Definition 5.2.8 (Basic Model Compatibility) *Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two modules. Let $AS^A(\mathcal{P}_1)$, respectively $AS^A(\mathcal{P}_2)$ be their annotated models. Let now $M_1^A \in AS^A(\mathcal{P}_1)$ and $M_2^A \in AS^A(\mathcal{P}_2)$ be two annotated models, they will be compatible if:*

$$Int(M_1^A) \cap At_v(\mathcal{P}_2) = Int(M_2^A) \cap At_v(\mathcal{P}_1)$$

▲

Now, for the second step of the cyclic compatibility criterion one takes models that passed the basic compatibility criterion and reconstruct their respective positive programs as defined previously. Then one computes the minimal model of the union of these reconstructed programs and see if the union of the originating models is a model of the union of their reconstructed programs (obtained through rules with form (5.1)).

Definition 5.2.9 (Annotation Compatibility) *Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two modules. Let $AS^A(\mathcal{P}_1)$, respectively $AS^A(\mathcal{P}_2)$ be their annotated models. Let now $M_1^A \in AS^A(\mathcal{P}_1)$ and $M_2^A \in AS^A(\mathcal{P}_2)$ be two basic compatible annotated models according to Definition 5.2.8. They will be compatible annotated models if:*

$$LM(P_{rec}(M_1^A) \cup P_{rec}(M_2^A) \cup trueInputs) = Int(M_1^A) \cup Int(M_2^A)$$

where

$$trueInputs = \left(Int(M_1^A) \bigcup Int(M_2^A)\right) \bigcap At_{in}(\mathcal{P}_1 \oplus \mathcal{P}_2)$$

▲

This compatibility check has the effect of blocking positive loops between modules that become self-supported once the modules are composed and relevant input atoms, previously giving support to rules involved in the loop, cease to exist. This is the second step we need towards a generalised module theorem, allowing us to retain compositionality in the face of positive loops.

5.2.3 Attaining Cyclic Compositionality

After setting the way by which one can deal with positive loops by using annotations in models, the join operator needs to be redefined. The original composition operators are applicable to annotated modules and this way, the positive dependencies of their atoms are added to their respective models.

Definition 5.2.10 (Modified Join) *Given two compatible annotated modules $\mathcal{P}_1, \mathcal{P}_2$ (in the sense of Definition 5.2.9), their composition is $\mathcal{P}_1 \otimes \mathcal{P}_2 = \mathcal{P}_1 \oplus \mathcal{P}_2$ provided that $\mathcal{P}_1 \oplus \mathcal{P}_2$ is defined.*

This way, given modules $\mathcal{P}_1$ and $\mathcal{P}_2$ and their annotated models, respectively M_1^A and M_2^A, their natural join, is defined as follows:

$$M_1^A \bowtie_A M_2^A = \left\{ ALM_{(P_{rec}(M_1^A) \cup P_{rec}(M_2^A)),\ At_{in}(\mathcal{P}_1 \oplus \mathcal{P}_2)} \mid \begin{array}{l} s.t.\ M_1^A \text{ and } M_2^A \text{ are} \\ \text{annotation compatible.} \end{array} \right\}$$

▲

We now follow by stating our main result in this Chapter.

Theorem 5.2.2 (Cyclic Module Theorem) *If $\mathcal{P}_1, \mathcal{P}_2$ are modules with annotated models such that $\mathcal{P}_1 \sqcup \mathcal{P}_2$ is defined, then:*

$$AS^A(\mathcal{P}_1 \oplus \mathcal{P}_2) = AS^A(\mathcal{P}_1) \bowtie_A AS^A(\mathcal{P}_2)$$

○

Proof of Theorem 5.2.2. Take $\mathcal{P}_1$ and $\mathcal{P}_2$. Let $M_1 \in AS(\mathcal{P}_1)$ and $M_2 \in AS(\mathcal{P}_2)$ and let $M \in AS(\mathcal{P}_1 \oplus \mathcal{P}_2)$ be an answer set of their union.

From Lemmas 6 and 7, we know that respectively M_1^A, M_2^A and M^A are their respective annotated answer sets.

Our annotated compatibility check in Definition 5.2.9 implies that in the subsequent Definition 5.2.10 the annotated join of models deriving from self-supported positive loops are disallowed, but preserves the rest of compatible models, including well supported loops.

Now, because the relaxed module theorem (Theorem 5.1.1) shows that the original MLP join and composition operators deal correctly with well supported positive loops, and because Definition 5.2.9 puts us now under its conditions, it is thus the case that

$$AS^A(\mathcal{P}_1 \oplus \mathcal{P}_2) = AS^A(\mathcal{P}_1) \bowtie_A AS^A(\mathcal{P}_2)$$

□

Theorem 5.2.2 extends the original module theorem with annotations, which allow us to detect self-supported loops. Considering two compatible annotated models M_1^A and M_2^A, by definitions of basic and annotation compatibility (Definitions 5.2.8 and 5.2.9), it is respectively the case that $Int(M_1^A) \cap At_v(\mathcal{P}_2) = Int(M_2^A) \cap At_v(\mathcal{P}_1)$ and

that $LM(P_{rec}(M_1^A) \cup P_{rec}(M_2^A) \cup trueInputs) = Int(M_1^A) \cup Int(M_2^A)$. The program reconstruction operation basically takes a model M^A and considers its annotations as providing support for literals, turning these into rules and constructing a positive program such that its single stable model trivially coincides with annotated model M^A from which it originated.

When assessing whether or not this model M^A is to be joined with some other model M'^A, we not only perform the compatibility checks that were originally prescribed in MLP and work for programs without self justifications, but also perform an extra verification to check whether or not the way the truth values of input atoms are varied to obtain the original MLP semantics actually introduces support to (self supported) positive loops. We thus do not include the set of facts which are no longer in the input signature of the composed module and would otherwise potentially provide support to self-justified loops but do add facts $trueInputs$ which guarantee that the behavior of the union of reconstructed programs is the same as the union of the original programs.

5.2.4 Shortcomings Revisited

Going back to Example 5.1.2, the new composition operator is able to produce the desired results:

Example 5.2.4 (Cyclic Dependencies Revisited) *Take again the two program modules in Example 5.1.2:*

$$\mathcal{P}_1 = \langle \{airbag \leftarrow safe.\}, \{safe\}, \{airbag\}, \emptyset \rangle$$
$$\mathcal{P}_2 = \langle \{safe \leftarrow airbag.\}, \{airbag\}, \{safe\}, \emptyset \rangle$$

which respectively have the following annotated models

$$AS^A(\mathcal{P}_1) = \{\{\}, \{airbag_{\{\{safe\}\}}, safe_{\{\{safe\}\}\}}\}$$
$$AS^A(\mathcal{P}_2) = \{\{\}, \{airbag_{\{\{airbag\}\}}, safe_{\{\{airbag\}\}}\}\}$$

while their composition and the (annotated) answer sets of their composition are:

$$\mathcal{P}_1 \otimes \mathcal{P}_2 = \langle \{airbag \leftarrow safe.\ \ safe \leftarrow airbag.\}, \{\}, \{safe, airbag\}, \emptyset \rangle$$

$$AS^A(\mathcal{P}_1 \otimes \mathcal{P}_2) = \{\{\}\}$$

Because of this, the annotated answer sets of their composition are the same as the join of the annotated answer sets of the individual modules:

$$AS^A(\mathcal{P}_1 \otimes \mathcal{P}_2) = AS^A(\mathcal{P}_1) \bowtie_A AS^A(\mathcal{P}_2)$$

■

5.3 Relation with Multi-Context Systems and their Compositionality

A Multi-Context System (MCS) Brewka & Eiter (2007b) is a collection of contexts that are linked using non-monotonic bridge rules. Each context has its own way of representing knowledge, including its own syntax and semantics. Non-monotonic bridge rules define how knowledge can be transferred between contexts. In MCSs, a model has the form of a collection of belief sets (called a belief state). An individual context is denoted as $M_{cs} := (L, kb, br)$ where L is its underlying logic, kb its knowledge base and br its bridge rules.

We overview next the semantics of MCSs that are of interest to us, which are as follows.

1. **Equilibrium semantics (ES)** defines intended models as exactly those belief states that, if viewed operationally, remain unchanged after first applying bridge rules and then applying contexts, hence the name of an equilibrium.

2. **Minimal equilibrium semantics (MES)** defines intended models as those equilibriums that are also minimal.

3. **Grounded equilibrium semantics (GES)** defines intended models as the minimal equilibriums of a positive MCS obtained by reducing the original MCS. Reducing MCSs is similar (both methodically and intent-wise) to the Gelfond/Lifschitz reduct operation Gelfond & Lifschitz (1988) use to define the stable model semantics.

4. **Supported Equilibrium Semantics (SES)**. Grounded equilibrium semantics (GES) is defined over MCSs in which all contexts are reducible. Thus, even one non-reducible context is enough to render GES non-applicable. Tasharrofi & Ternovska (2014) solved this by proposing an intermediate semantics (SES) capturing the robustness of ES, thus making it applicable to every MSC while — like GES does — dealing with the problem of self justified loops.

5.3.1 Supported Equilibrium Semantics (SES)

We present next an overview of the supported equilibrium semantics Tasharrofi & Ternovska (2014).

5.3.1.1 Support for Contexts

Justifications are intuitively introduced as: "Justifications for belief set b_s are possible explanation of why beliefs in b_s are believed. Then, also intuitively, justifications are used to define the support for a logic: $Sup_L(k_b, b_s)$ denotes a (usually non-exhaustive) set of possible justifications for b_s.

Using the definition of supports on the level of logic, we present next their definition of support on the level of contexts. Unlike supports for logics that worked with syntactic (knowledge base formulas) and semantic (beliefs) objects simultaneously, in Definition 5.3.1, supports at the level of contexts work solely on semantic objects (beliefs).

Definition 5.3.1 (Support for Contexts Tasharrofi & Ternovska (2014)) *Consider MCS* $M := (C_1, \ldots, C_n)$*, its context* $Ci := \langle L_i, kb_i, br_i \rangle$*, and its belief state* $S := (S_1, \ldots, S_n)$*. Also, let* Sup_{L_i} *be the support function for logic* L_i*.*

Support of belief set S_i *(from context* C_i *) under* S*, denoted by* $Sups_{S_i}$*, is the set of functions*[2] $f : S_i \to P(X(S))$ *that are computed by taking a function*[3] $g \in Sup_{L_i}(kb_i \cup app_i(S), S_i)$ *and tracing the reason for the inclusion of the knowledge that comes from bridge rules. More formally, a function* f *is included in* $Sups_{S_i}$ *if and only if functions* $g \in Sup_{L_i}(kb_i \cup app_i(S), S_i)$ *and* $R : S_i \to P(br_i)$ *exist such that, for all* $b \in S_i$*, we have*[4]*:*

$$f(b) := \{i : b' | b' \in fst(g(b))\} \bigcup_{r \in R(b)} Body^+(r).$$

and $R(b) \subseteq \{r \mid r \in br_i \text{ and } S \models Body(r)\}$ *is a minimal subset of applicable bridge rules that justifies the knowledge that comes from bridge rules, i.e., for all formulas* $k \in (snd(g(b)) \backslash kb)$*, a rule* $r \in R(b)$ *exists with* $Head(r) = k$*.* ▲

Note that, in Definition 5.3.1, if $snd(g(b)) \backslash kb = \emptyset$ for some belief b, then $R(b) = \emptyset$. Intuitively, it means that support from other contexts is required only when existing knowledge of a context is not sufficient for supporting a belief.

Supports at the level of contexts are used to define supported equilibrium semantics for MCSs Tasharrofi & Ternovska (2014). In the following definition, the authors provide the notion of a supported equilibrium. Informally, a belief state is called a supported equilibrium if all beliefs are well-justified, i.e., they are justified and nothing justifies itself (either directly or indirectly). Self-justifications are avoided by requiring the existence of a well-ordering on the beliefs $X(S)$.

Definition 5.3.2 (Supported Equilibrium Tasharrofi & Ternovska (2014)) *A belief state* $S := (S_1, \ldots, S_n)$ *of MCS is a supported equilibrium with respect to* $(Sup_{L_1}, \ldots, Sup_{L_n})$ *if functions* $f_1 \in Sups_{S_1}, \ldots, f_n \in Sups_{S_n}$ *and well-founded strict partial ordering* $<$ *on* $X(S)$ *exist s.t. if* $p \in S_i$ *and* $(j : q) \in f_i(p)$ *then* $(j : q) < (i : p)$*.* ▲

First, note that Definition 5.3.2 does not put any special requirement on contexts and works for all contexts and all supports. Therefore, unlike grounded equilibrium semantics of Brewka & Eiter (2007b), introspection in supported equilibrium semantics

[2] Where $X(S)$ is used for the disjoint union of beliefs in all belief states S_i, i.e., $X(S) := \{(i : b) \mid 1 \leq i \leq n \text{ and } b \in S_i\}$.

[3] Where $app_i(S) = \{Head(r) \mid r \in br_i \wedge S \models Body(r)\}$ is used to denote the heads of all applicable bridge rules of context C_i wrt. S.

[4] They use, for a pair $P := (X, Y)$, $fst(P)$ to denote X and $snd(P)$ to denote Y.

does not come at the cost of excluding non-reducible contexts. Second, they note that Definition 5.3.2 tests a belief state for being supported but not for being an equilibrium. So, one might reasonably suspect that a belief state S might exist such that S is a supported equilibrium (according to Definition 5.3.2) but not an equilibrium (according to the original definition of equilibrium semantics Brewka & Eiter (2007b). However, their Theorem 1 states that if S is a supported equilibrium then it has to be an equilibrium as well. Therefore, the term "supported equilibrium" is a reasonable name for belief states that satisfy the condition of Definition 5.3.2.

5.3.2 Defining a Notion of Compositionality for Multi-Context Systems

As for the notion of compositionality we presented at the very beginning of this document, MCS (even under the supported equilibrium semantics) are not compositional in the sense of compositionality in MLP.

It is the case though that MLPs are translatable to MCSs where contexts have the, so called in Tasharrofi & Ternovska (2014), "*logic of normal answer set programs under stable model semantics*" which we will henceforth denote as *Lasp*. In this case, and because in MLP it is the case that visible atoms are globally available and usable by other modules, we define their translation to multi-context systems as follows:

Definition 5.3.3 (Translation of MLP to MCS) *Given n MLP modules $M^1_{lp},\ldots,M^n_{lp}$, we obtain from it a multi-context context system formed by contexts*

$$M^1_{cs} := (L_1, kb_1, br_1), \ldots, M^n_{cs} := (L_n, kb_n, br_n)$$

such that logic $L_1 = \ldots = L_n = Lasp$, bridge rules $r \in br_j$ will be all rules of the following type:

$$a \leftarrow C_k : a.\ s.t.\ \ j,k \in \{1,\ldots,n\} \tag{5.3}$$

such that a is an input atom in some MLP module, namely $a \in At_{in}(M^j_{lp})$ and a is an output atom in some other MLP module, namely $a \in At_{out}(M^k_{lp})$. Finally $kb_j = Rules(M^j_{lp})$. ▲

Note that in the presence of a new context, the corresponding bridge rules must be created between contexts being intended for combination. Having these rules, new models must then calculated for the system.

MCSs having the logic *Lasp* are also translatable to MLPs by constructing a module for the bridge rules associated to a context and another module for the rules of the context.

Definition 5.3.4 (Translation of MCS to MLP) *Consider a reducible context $C := (L, kb, br)$ where $L = Lasp$ is its underlying logic, kb is its knowledge base and br its set of non-monotonic bridge rules. A translation to an MLP setting is defined as follows:*

$$M_i = \langle P_i, I_i, At_C, \emptyset \rangle$$

Where P_i contains, for each rule r of the following type s.t. $r \in kb$:

$$L_0 \leftarrow L_1, \ldots, L_m, not\ L_{m+1}, \ldots, not\ {}_{L_n}.$$

a rule r' of the following (translated) type:

$$C_i : L_0 \leftarrow C_i : L_1, \ldots, C_i : L_m, not\ C_i : L_{m+1}, \ldots, not\ C_i : L_n.$$

together with rules r of the following type s.t. $L_0 \leftarrow Body(r). \in br_i$

$$C_i : L_0 \leftarrow Body(r).$$

Furthermore, the input signature of I_i of M_i is:

$$I_i = \{C_i : L_i \mid C_i : L_i \in Body(r).\ s.t.\ r \in br_i\}$$

and the output signature At_C of M_i is:

$$At_C = \{C_i : L \mid L \text{ is an atom in the language of context } C_i\}$$

▲

All MCS in this class are reducible and as such SES-semantics is not necessary as we could simply apply the ES-Semantics. However, consider now the following Example 5.3.1 taken from Tasharrofi & Ternovska (2014).

Example 5.3.1 *Let $M := (C_1, C_2, C_3)$ be*[5] *a multi-context system with $C_i := (L_i, kb_i, br_i)$ $(for\ i \in \{1,2,3\})$, L_i be the logic of normal answer set programs under stable model semantics and kb_i, br_i be as follows:*

$$C_1 := \begin{cases} kb_1 := \{ & forbes400(bg). \\ & wealthy(X) \leftarrow forbes400(X). \\ & wealthy(X) \leftarrow celebrity(X). \} \\ \\ br_1 := \{ & celebrity(X) \leftarrow C2 : famous(X). \} \end{cases}$$

$$C_2 := \begin{cases} kb_2 := \{ & actor(bp).\ famous(X) \leftarrow actor(X). \} \\ \\ br_2 := \{ & famous(X) \leftarrow C_1 : wealthy(X). \\ & famous(X) \leftarrow C_3 : comedian(X). \} \end{cases}$$

$$C_3 := \begin{cases} kb_3 := \{ & \{comedian(jk).\} \} \\ \\ br_3 := \{ & \bot \leftarrow C_3 : comedian(X),\ not\ C_2 : famous(X). \} \end{cases}$$

[5] The authors use := to represent "denotes" or "equals by definition".

*With X ranging over the possibilities, namely "bg" (Bill Gates), "bp" (Brad Pitt), "jk" (Jimmy Kimmel) and "aj" (Average Joe). According to the equilibrium semantics, M has equilibriums S*1,...,*S*6 *as follows:*

$$\textit{for } i \in \{1,\ldots,6\} : Si := (bs_1^i, bs_2^i, bs_3^i) \textit{ where,}$$

$$\begin{array}{rl}
bs_1^1 := & \{forbes400(bg), wealthy(bg), wealthy(bp), \\
& celebrity(bg), celebrity(bp)\} \\
bs_1^2 := & bs_1^1 \cup \{wealthy(aj), celebrity(aj)\}, \\
bs_1^3 := & bs_1^4 := bs_1^1 \cup \{wealthy(jk), celebrity(jk)\}, \\
bs_1^5 := & bs_1^6 := bs_1^2 \cup bs_1^3, \\
bs_2^1 := & \{actor(bp), famous(bg), famous(bp)\}, \\
bs_2^2 := & bs_2^1 \cup \{famous(aj)\}, \\
bs_2^3 := & bs_2^4 := bs_2^1 \cup \{famous(jk)\}, \\
bs_2^5 := & bs_2^6 := bs_2^2 \cup bs_2^3, \\
bs_3^1 := & bs_3^2 := bs_3^3 := bs_3^5 := \{\}, \\
bs_3^4 := & bs_3^6 := \{comedian(jk)\}.
\end{array}$$

From which only one is a grounded equilibrium:

$$S_1 := \left(\begin{array}{ll}
\{ & forbes400(bg), wealthy(bg), wealthy(bp), \\
& celebrity(bg), celebrity(bp)\}, \\
\{ & actor(bp), famous(bg), famous(bp)\}, \\
\{ & \}.
\end{array} \right)$$

If the names in this example are to be taken literally, among all the six equilibrium models above, only S_4 is a reasonable belief state. This is because "Average Joe" by definition is not famous, wealthy, or a celebrity, and also because "Jimmy Kimmel" is a famous comedian and a wealthy celebrity.

The two answer sets of the MLP translation correspond to having or not an atom $comedian(jk)$, from which AS_2 corresponds exactly to S_1 and AS_1 corresponds to our intended belief state S_4:

$$AS_1 : \left\{ \begin{array}{l}
actor(bp), forbes400(bg), comedian(jk), wealthy(bp), wealthy(bg), \\
wealthy(jk), celebrity(bp), celebrity(bg), celebrity(jk), \\
famous(bp), famous(bg), famous(jk)
\end{array} \right\}$$

$$AS_2 : \left\{ \begin{array}{l}
actor(bp), forbes400(bg), wealthy(bp), wealthy(bg), celebrity(bp), \\
celebrity(bg), famous(bp), famous(bg)
\end{array} \right\}$$

△

The supported equilibrium semantics (SES) coincides in this case with the answer sets of the union of bridge rules and knowledge bases within contexts under composition which, as we have seen, are the answer sets of our generalised MLP models

and we conjecture that it is indeed applicable to solving the positive cyclic dependency issue but leave this for future work. No compositionality results (in the spirit of the module theorem) are provided in the literature though and even if the supported equilibrium semantics provides a reasonable way to evaluate MCSs under the stable model semantics, compositionality is not achieved. This conclusion follows from **(1)** our result showing that models must be richer than simply answer sets in terms of the information they contain about support, in order to allow for compositionality in the sense of MLP's module theorem, and **(2)** the fact that a suitable join operator for models is not provided.

Relation to our Dependency Annotations Supported equilibrium semantics is defined in terms of support functions $f(b)$ that take support at the level of contexts for a literal b as knowledge coming from the positive part of the bodies of applicable bridge rules. This is intuitively very similar to the fact that in our generalised MLP approach we take dependency annotations as being positive atoms in the bodies of applicable rules having a given target literal, for which we want to track the dependency, in their head. We furthermore only consider (chains of) dependencies stemming from input atoms of a given module which as we have seen are translated into MCS as bridge rules. We present next a conjecture of a correspondence between multi-context systems and modular logic programs, and follow-up with a discussion of a possible path for proving it. We leave this as a partial open-end for future work.

Conjecture 5.3.1 (Correspondence of MCS and MLP) *Let M_{cs} be a reducible multi-context system under Lasp and let M_{lp} be its translation to MLP according to Definition 5.3.4. The SES-models of M_{cs} correspond to the answer sets of M_{lp}. The converse, using the inverse translation from MLPs to MCSs in Definition 5.3.3, is also true.*

Let M_{cs} be a reducible multi-context system under *Lasp* and let M_{lp} be its translation to MLP by Definition 5.3.4.

For the $(\rightarrow)$ direction, one can start by assuming, towards a contradiction, that there is a supported equilibrium S_{eq} of M_{cs} s.t. $S_{eq} \notin AS(M_{lp})$. Then, either S_{eq} is not a model of the reduct $M_{lp}^{S_{eq}}$ or that S_{eq} is not a minimal model.

By Definition 5.3.2 of supported equilibria, functions $f_1 \in Sup_{S_1}, \ldots, f_n \in Sup_{S_n}$ and a well-founded strict partial ordering $<$ on $X(S)$ exist s.t. if $p \in S_i$ and $(j:q) \in f_i(p)$ then $(j:q) < (i:p)$. This means respectively that all atoms in S_{eq} are supported and that no support comes from self-supported loops. Now, by Definition 5.2.4, an annotated model of M_{lp} is the least fixed point $lfp(T^A_{P,In})$ which implicitly imposes an ordering similar to the well-founded strict partial ordering $<$ on $X(S)$ imposed in S_{eq}. This contradicts our assumption that S_{eq} is not a minimum model of M_{lp}. The direct way in which the rules are translated by Definition 5.3.4, should lead by fixed-point induction to S_{eq} being a model of M_{lp} which would contradict our assumption that S_{eq} is not a model of M_{lp}.

As for the converse direction ($\leftarrow$), using Definition 5.3.3, the proof should be straightforward by fixed-point induction.

5.4 Conclusions and Future Work

We lift the restriction that disallows composing modules with cyclic dependencies in the framework of Modular Logic Programming Oikarinen & Janhunen (2008). We present a model join operation that requires one to look at every model of two modules being composed in order to check for minimality of models that are comparable on account of their inputs. This operation is able to distinguish between atoms that are self supported through positive loops and atoms with proper support, allowing one to lift the condition disallowing positive dependencies between modules. However, this approach is not local as it requires comparing every models and, as it is not general because it is restricted to positive programs.

Because of this lack of generality of the former approach, we present an alternative solution requiring the introduction of extra information in the models for one to be able to detect dependencies. We use models annotated with the way they depend on the atoms in their module's input signature. We then define their semantics in terms of a fixed point operator. After setting the way by which one deals with positive loops by using annotations in models, the join operator needs to be redefined. The original composition operators are applicable to annotated modules. This way, their positive dependencies are added to their respective models. This approach turns out to be local, in the sense that we need only look at two models being joined and unlike the first alternative we presented, it is applicable to all programs. This is the most important contribution of this chapter.

We also provide, as far as we know, the first formal definition of a connection between MLP with reducible multi-context systems by combining the approaches of Chapters 4 and 5 we obtain a fully compositional semantics for modular logic programs.

As we have seen before, Dao-Tran *et al.* (2009) provide an embedding of the original composition operator of Oikarinen and Janhunen into their approach. Since our constructions in Chapter 4 rely on a transformational approach using operator $\sqcup$ of Oikarinen and Janhunen, by composing both translations, an embedding into Dao-Tran *et al.* (2009) is immediately obtained. It remains to be checked whether the same translation can be used in the presence of positive cycles.

Chapter 6

Modular P-Log: A Probabilistic Extension to Modular Logic Programming

In this chapter we propose and discuss an approach for modularising P-log programs and corresponding compositional semantics based on conditional probability measures. We do so by resorting to Oikarinen and Janhunen's definition of a logic program module and extending it to P-log by introducing the notions of input random attributes and output literals. For answering to P-log queries our method does not imply calculating all the answer sets (possible worlds) of a given program, and previous calculations can be reused. Our framework also handles probabilistic evidence by conditioning (observations).

6.1 Introduction and Motivation

The P-log language Baral *et al.* (2004) has emerged as one of the most flexible frameworks for combining probabilistic reasoning with logical reasoning, in particular, by distinguishing acting (doing) from observations and allowing non-trivial conditioning forms Baral *et al.* (2004); Baral & Hunsaker (2007). It is a non-monotonic probabilistic logic language supported by two major formalisms, namely Answer Set Programming Gelfond & Lifschitz (1988, 1990); Lifschitz (2008a,b) for declarative knowledge representation and Causal Bayesian Networks Pearl (2000) as its probabilistic foundation. In particular, ordinary Bayesian Networks can be encoded in P-log. The relationships of P-log to other alternative uncertainty knowledge representation languages like Kwiatkowska *et al.* (2001); Pfeffer & Koller (2000); Poole (1997) have been studied in Baral *et al.* (2004). Unfortunately, the existing current implementations of P-log Anh *et al.* (2008); Gelfond *et al.* (2006) have exponential best case complexity, since they enumerate all possible models, even though it is known that for singly connected Bayesian Networks (polytrees) reasoning can be performed in

polynomial time Pearl (1988).

The contributions of this chapter are the definition of compositional modules for the P-log language, their corresponding compositional semantics as well as the definition of their probabilistic interpretation. Our semantics relies on a translation to the aforementioned logic program modules in Oikarinen & Janhunen (2008). With this appropriate notion of P-log modules one can obtain possible worlds incrementally, and this can be optimised for answering to probabilistic queries in polynomial time for specific cases, using techniques inspired in the variable elimination algorithm proposed in Zhang & Poole (1996).

The rest of this chapter is organised as follows. Section 6.2 briefly overviews P-log syntax and semantics as well as recalls the essential modularity results for answer set programming. Next, Section 6.3 is the core of this chapter defining modules for P-log language as well as its translation into (G)MLP modules. The subsequent section presents the module theorem and a discussion of the application of the result to Bayesian Networks. We conclude with final remarks and foreseen work.

6.2 Preliminaries

In this section, we start by reviewing the syntax and semantics of the P-log language Baral *et al.* (2004), and illustrate it with an example encoding a Bayesian Network. The reader is assumed to have familiarity with (Causal) Bayesian Networks Pearl (2000). A good introduction to Bayesian Networks can be found in Russell & Norvig (2010).

6.2.1 P-log Programs

P-log is a declarative language Baral *et al.* (2004), based on a logic formalism for probabilistic reasoning and action, that uses answer set programming (ASP) as its logical foundation and Causal Bayesian Networks (CBNs) as its probabilistic foundation. P-log is a complex language to present and the reader is referred to Baral *et al.* (2004) for full details. We try to make its presentation self-contained for this chapter, abbreviating or even neglecting the irrelevant parts, and following closely the way it is presented in Baral *et al.* (2004).

6.2.1.1 P-log syntax.

A probabilistic logic program (P-log program) Π consists of

Definition 6.2.1 (Probabilistic Logic Program (P-log) Π)

(i) a sorted signature,

(ii) a declaration part,

(iii) a regular part,

(iv) a set of random selection rules,

(v) a probabilistic information part, and

(vi) a set of observations and actions.

Notice that the first four parts correspond to the generation of actual answer sets, and the last two define the probabilistic information.

The **declaration part** defines a sort c by explicitly listing its members with a statement $c = \{x_1, \ldots, x_n\}$, or by defining a unary predicate c in a program with a single answer set. An attribute a with n parameters is declared by a statement $a : c_1 \times \ldots \times c_n \rightarrow c_0$ where each c_i is a sort $(0 \leq i \leq m)$; in the case of an attribute with no parameter the syntax $a : c_0$ may be used. By $range(a)$ we denote the set of elements of sort c_0. The sorts can be understood as domain declarations for predicates and attributes used in the program, for appropriate typing of argument variables.

The **regular part** of a P-log program consists in a set of Answer Set Programming rules (without disjunction) constructed from the usual ASP literals plus attribute literals of the form $a(\bar{t}) = t_0$ (including strongly negated literals), where $\bar{t}$ is a vector of n terms and t_0 is a term, respecting the corresponding sorts in the attribute declaration. Given a sorted signature Σ we denote by $Lit(\Sigma)$ the set of literals in Σ (i.e., Σ-literals) excluding all unary atoms $c_i/1$ used for specifying sorts.

Random selection rules define random attributes and possible values for them through statements of the form:

$$[r]\ random(a(\bar{t}) : \{X : p(X)\}) \leftarrow B.$$

expressing that if B holds then the value of $a(\bar{t})$ is selected at random from the set $\{X : p(X)\} \cap range(a)$ by experiment r, unless this value is fixed by a deliberate action, with r being a term uniquely identifying the rule. The concrete probability distribution for random attributes is conveyed by the probabilistic information part containing *pr-atoms* (probability atoms), of the form:

$$pr_r(a(\bar{t}) = y|_c B) = v$$

stating that if the value of $a(\bar{t})$ is fixed by experiment r and B holds, then the probability that r causes $a(\bar{t})$ to take value y is v, with $v \in [0,1]$. The condition B is a conjunction of literals or the default negation (*not*) of literals.

Finally, **observations** and **actions** are statements of the form $obs(l)$ and $do(a(\bar{t}) = y)$, respectively, where l is an arbitrary literal of the signature.

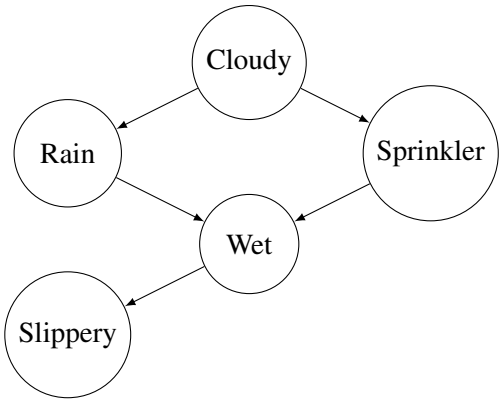
Example 6.2.1 (Wet Grass) *Suppose that there are two events which could cause grass to be wet: either the sprinklers are on, or it is raining. Furthermore, suppose that the rain has a direct effect on the use of the sprinklers (namely that when it rains, the sprinklers are usually not turned on). Furthermore, cloudy sky affects whether the sprinklers are on and obviously if it is raining or not. Finally, notice that, the grass being wet or dry affects it being slippery or not.*

This scenario can be modeled with a Bayesian network (shown in Figures 6.1 and 6.2). All random variables are Boolean and have no parameters; also notice

how the conditional probability tables (CPTs) are encoded with pr-atoms as well as causal dependencies among random attributes. The P-log semantics will take care of completing the CPTs assuming a uniform distribution for the remaining attribute values (e.g., $cloudy = false$ will have probability 0.5). Rules can be used to extract additional knowledge from the random variables (e.g., the "dangerous if slippery" rule). In particular we will be able to query the program to determine the probability $\mathbf{P}(dangerous|sprinkler = t)$.

$Boolean = \{t, f\}.$
$cloudy : Boolean.$
$rain : Boolean.$
$sprinkler : Boolean.$
$wet : Boolean.$
$slippery : Boolean.$

$dangerous \leftarrow slippery = t.$



$[rc]\, random(cloudy, \{X : Boolean(X)\}).$
$[rr]\, random(rain, \{X : Boolean(X)\}).$
$[rsk]\, random(sprinkler, \{X : Boolean(X)\}).$
$[rw]\, random(wet, \{X : Boolean(X)\}).$
$[rsl]\, random(slippery, \{X : Boolean(X)\}).$

Figure 6.1: Bayesian Network encoded in P-log

△

6.2.1.2 P-log semantics.

The semantics of a P-log program Π is given by a collection of the possible sets of beliefs of a rational agent associated with Π, together with their probabilities. We refer to these sets of beliefs as possible worlds of Π. Note that due to the restriction on the signature of P-log programs the authors enforce (all sorts are finite), possible worlds of Π are always finite. The semantics is defined in two stages. First we will define a mapping of the logical part of Π into its Answer Set Programming counterpart, $\tau(\Pi)$. The answer sets of $\tau(\Pi)$ will play the role of possible worlds of Π. The probabilistic part of Π is used to define a measure over the possible worlds, and from these the probabilities of formulas can be determined. The set of all possible worlds of Π will

$$pr_{rr}(rain = t \mid_c cloudy = f) = 0.2.$$
$$pr_{rr}(rain = t \mid_c cloudy = t) = 0.8.$$

$$pr_{rsk}(sprinkler = t \mid_c cloudy = f) = 0.5.$$
$$pr_{rsk}(sprinkler = t \mid_c cloudy = t) = 0.1.$$

$$pr_{rsl}(slippery = t \mid_c wet = f) = 0.1.$$
$$pr_{rsl}(slippery = t \mid_c wet = t) = 0.9.$$

$$pr_{rw}(wet = t \mid_c sprinkler = f, rain = f) = 0.0.$$
$$pr_{rw}(wet = t \mid_c sprinkler = f, rain = t) = 0.9.$$
$$pr_{rw}(wet = t \mid_c sprinkler = t, rain = f) = 0.9.$$
$$pr_{rw}(wet = t \mid_c sprinkler = t, rain = t) = 0.99.$$
$$obs(sprinkler = t).$$

Figure 6.2: Conditional Probability Tables for CBN in Figure 6.1

be denoted by $\Omega(\Pi)$.

The Answer Set Program $\tau(\Pi)$ is defined in the following way, where capital letters are variables being grounded with values from the appropriate sort. To reduce overhead we omit the sort predicates for variables in the program rules. It is also assumed that any attribute literal $a(\bar{t}) = y$ is replaced consistently by the predicate $a(\bar{t}, y)$ in the translated program $\tau(\Pi)$, constructed as follows:

τ_1: For every sort $c = \{x_1, \ldots, x_n\}$ of Π, $\tau(\Pi)$ contains facts $c(x_1), \ldots, c(x_n)$. For any remaining sorts defined by an ASP program T in Π, then $T \subseteq \tau(\Pi)$.

τ_2: Regular part:

- **(a)** For each rule r in the regular part of Π, $\tau(\Pi)$ contains the rule obtained by replacing each occurrence of an atom $a(\bar{t}) = y$ in r by $a(\bar{t}, y)$.
- **(b)** For each attribute term $a(\bar{t})$, $\tau(\Pi)$ contains $\neg a(\bar{t}, Y_1) \leftarrow a(\bar{t}, Y_2), Y_1 \neq Y_2$ guaranteeing that in each answer set $a(\bar{t})$ has at most one value.

τ_3: Random selections:

- **(a)** For an attribute $a(\bar{t})$, we have the rule: $intervene(a(\bar{t})) \leftarrow do(a(\bar{t}, Y))$. Intuitively, the value of $a(\bar{t})$ is fixed by a deliberate action, i.e., $a(\bar{t})$ will not be considered random in possible worlds satisfying $intervene(a(\bar{t}))$.
- **(b)** Random selection $[r]$ $random(a(\bar{t}) : \{X : p(X)\}) \leftarrow B$ is translated into rule $1\{\ a(\bar{t}, Z) : poss(r, a(\bar{t}), Z)\ \}1 \leftarrow B$, not $intervene(a(\bar{t}))$. and $poss(r, a(\bar{t}), Z) \leftarrow c_0(Z),\ p(Z), B$, not $intervene(a(\bar{t}))$. with $range(a) = c_0$.

τ_4: Each pr-atom $pr_r(a(\bar{t}) = y|_c\ B) = v$ is translated into the following rule $pa(r,a(\bar{t},y),v) \leftarrow poss(r,a(\bar{t}),y),B$ of $\tau(\Pi)$ with $pa/3$ a reserved predicate.

τ_5: $\tau(\Pi)$ contains actions and observations of Π.

τ_6: For each Σ-literal l , $\tau(\Pi)$ contains the constraint $\leftarrow obs(l), not\ \ l$.

τ_7: For each atom $a(t) = y$, $\tau(\Pi)$ contains the rule $a(\bar{t},y) \leftarrow do(a(\bar{t},y))$.

In the previous construction, the two last rules guarantee respectively that no possible world of the program fails to satisfy observation l, and that the atoms made true by the action are indeed true. The introduction of reserved predicates $poss/3$ and $pa/3$ is a novel contribution we make to the transformation, and has the purpose of simplifying the presentation of the remaining details of the semantics.

P-log semantics assigns a probability measure for each world W, i.e., answer set, of $\tau(\Pi)$ from the causal probability computed deterministically from instances of predicates $poss/3$ and $pa/3$ true in the world. Briefly, if an atom $pa(r,a(\bar{t},y),v)$ belongs to W then the causal probability $\mathbf{P}(W,a(\bar{t}) = y)$ is v, i.e., the assigned probability in the model. The possible values for $a(\bar{t})$ are collected by $poss(r,a(\bar{t},y_k))$ instances true in W, and P-log semantics assigns a (default) causal probability for non-assigned values, by distributing uniformly the non-assigned probability among these non-assigned values.

The details to make this formally precise are rather long Baral *et al.* (2004) but for our purpose it is enough to understand that for each world W the causal probability $\sum_{y \in range(a)} \mathbf{P}(W,a(\bar{t}) = y) = 1.0$, for each attribute term with at least a possible value. These probability calculations can be encoded in ASP by means of aggregate rules (#*sum* and #*count*), making use of only $pa/3$ and $poss/3$ predicates.

Example 6.2.2 *Consider the P-log program of Example 6.2.1. This program has 16 possible worlds (notice that $sprinkler = t$ is observed and thus fixed). One possible world is W_1 containing:*

$cloudy(f)$	$\neg cloudy(t)$
$rain(f)$	$\neg rain(t)$
$wet(t)$	$\neg wet(f)$
$sprinkler(t)$	$\neg sprinkler(f)$
$slippery(t)$	$\neg slippery(f)$

$dangerous$
$obs(sprinkler(t))$

Furthermore the following probability assignment atoms are true in that model:

$poss(rc,cloudy,t)$	$poss(rc,cloudy,f)$	
$poss(rr,rain,t)$	$poss(rr,rain,f)$	$pa(rr,rain(t),0.2)$
$poss(rsk,sprinkler,t)$	$poss(rsk,sprinkler,f)$	$pa(rsk,sprinkler(t),0.5)$
$poss(rsl,slippery,t)$	$poss(rsl,slippery,f)$	$pa(rsl,slippery(t),0.9)$
$poss(rw,wet,t)$	$poss(rw,wet,f)$	$pa(rw,wet(t),0.9)$

which determines the following causal probabilities in the model

$$\begin{array}{ll} \boldsymbol{P}(W_1, cloudy = t) = \frac{1.0-0.0}{2} = 0.5 & \boldsymbol{P}(W_1, cloudy = f) = \frac{1.0-0.0}{2} = 0.5 \\ \boldsymbol{P}(W_1, rain = t) = 0.2 & \boldsymbol{P}(W_1, rain = f) = \frac{1.0-0.2}{1} = 0.8 \\ \boldsymbol{P}(W_1, sprinkler = t) = 0.5 & \boldsymbol{P}(W_1, sprinkler = f) = \frac{1.0-0.5}{1} = 0.5 \\ \boldsymbol{P}(W_1, wet = t) = 0.9 & \boldsymbol{P}(W_1, wet = f) = \frac{1.0-0.9}{1} = 0.1 \\ \boldsymbol{P}(W_1, slippery = t) = 0.9 & \boldsymbol{P}(W_1, slippery = f) = \frac{1.0-0.9}{1} = 0.1 \end{array}$$

△

The authors define next the measure μ_Π induced by a P-log program Π:

Definition 6.2.2 (Measure) *Let W be a possible world of a P-log program Π. The unnormalised probability of W induced by Π is*

$$\hat{\mu}_\Pi(W) = \prod_{a(\bar{t},y)\in W} \boldsymbol{P}(W, a(\bar{t}) = y)$$

where the product is taken over atoms for which $\boldsymbol{P}(W, a(\bar{t}) = y)$ is defined.

If Π is a P-log program having at least one possible world with nonzero unnormalised probability, then the measure, $\mu_\Pi(W)$, of a possible world W induced by Π is the normalised probability of W divided by the sum of the unnormalised probabilities of all possible worlds of Π, i.e.,

$$\mu_\Pi(W) = \frac{\hat{\mu}_\Pi(W)}{\sum_{W_i\in\Omega} \hat{\mu}_\Pi(W_i)}.$$

When the program Π is clear from the context we may simply write $\hat{\mu}$ and μ instead of $\hat{\mu}_\Pi$ and μ_Π respectively. ▲

Example 6.2.3 *For world W_1 of Example 6.2.2 we obtain that:*

$$\begin{array}{rcl} \hat{\mu}(W_1) & = & \boldsymbol{P}(W_1, cloudy = f) \times \boldsymbol{P}(W_1, rain = f) \times \boldsymbol{P}(W_1, wet = t) \times \\ & & \boldsymbol{P}(W_1, sprinkler = t) \times \boldsymbol{P}(W_1, slippery = t) = \\ & = & 0.5 \times 0.8 \times 0.9 \times 0.5 \times 0.1 = 0.018 \end{array}$$

Since the sum of the unconditional probability measure of all the sixteen worlds of the P-log program is 0.3, then we obtain that the normalised probability for this possible world is $\mu(W_1) = 0.06$. △

The truth and falsity of propositional formulas with respect to possible worlds are defined in the standard way. A formula A, true in W, is denoted by $W \vdash A$.

Definition 6.2.3 (Probability) *Suppose Π is a P-log program having at least one possible world with nonzero unnormalised probability. The probability, $P_\Pi(A)$, of a formula A is the sum of the measures of the possible worlds of Π on which A is true, i.e.:*

$$P_\Pi(A) = \sum_{W\vdash A} \mu_\Pi(W).$$

▲

Conditional probability in P-log is defined in the usual way by $P_{\Pi}(A|B) = P_{\Pi}(A \wedge B)/P_{\Pi}(B)$ whenever $P_{\Pi}(B) \neq 0$, where the set B stands for the conjunction of its elements. Moreover, under certain consistency conditions on P-log programs Π, formulas A, and a set of literals B such that $P_{\Pi}(B) \neq 0$, it is the case that $P_{\Pi}(A|B) = P_{\Pi \cup obs(B)}(A)$. See the original work Baral *et al.* (2004) where the exact consistency conditions are stated, which are assumed to hold subsequently.

Example 6.2.4 *Continuing Example 6.2.3, consider now* $P(dangerous \mid sprinkler = t,\ wet = t)$. *Since* $dangerous \leftarrow slippery = t$ *we are interested only on models where* $slippery = t$. *The four possible worlds and their respective probabilities are thus:*

$$\begin{aligned}
\{sp(t), sl(t), d, w(t), r(t), c(t)\} &= 0.3564 \\
\{sp(t), sl(t), d, w(t), r(t), c(f)\} &= 0.3564 \\
\{sp(t), sl(t), d, w(t), r(f), c(t)\} &= 0.0891 \\
\{sp(t), sl(t), d, w(t), r(f), c(f)\} &= 0.0891
\end{aligned}$$

Considering the obvious abbreviations. △

6.2.2 Related Work

The PRISM system Sato (1995) is a general programming language with learning ability for statistical parameters embedded in programs which allows building complex statistical models. The theoretical background of PRISM is distribution semantics for parameterised logic programs and EM (Expectation-Maximisation) learning of their parameters from observations (cf. Sato (1995); Sato & Kameya (2001)).

Being comprised of two subsystems, one for learning and the other for execution, a program in the PRISM system is a logic program in which facts have a parametrised probability distribution so that the program can be seen as a parameterised statistical model. The program defines a probability distribution (probability measure) over the set of possible Herbrand interpretations. In execution, probabilities or samples of various probabilistic constructs in the program will be calculated according to the defined distribution. In learning, the authors use ML (maximum likelihood) estimation of the program parameters from incomplete data by the EM algorithm so that the defined distribution is closer to the observed distribution. Because PRISM programs can be arbitrarily complex, their model can be arbitrarily complex as well, which means it is possible to build large yet understandable symbolic-statistical models that go beyond traditional statistical models.

P-log and PRISM Sato (1995) share a substantial number of common features. Both are declarative languages capable of representing and reasoning with logical and probabilistic knowledge. In both cases, logical part of the language is rooted in logic programming. There are also substantial differences. The PRISM system seems to be primarily intended as "a powerful tool for building complex statistical models" with emphasis of using these models for statistical learning. As a result PRISM allows infinite possible worlds, and has the ability of learning statistical parameters embedded

in its inference mechanism. The goal of P-log designers was to develop a knowledge representation language allowing natural, elaboration tolerant representation of common sense knowledge involving logic and probabilities. Infinite possible worlds and algorithms for statistical learning were not a priority. Instead the emphasis was on greater logical power provided by Answer Set Prolog, on causal interpretation of probability, and on the ability to perform and differentiate between various types of updates.

We have so far discussed logic programming approaches that attempt to integrate logical and probabilistic reasoning. Apart from these, Vos & Vermeir (2000) presented a notion where the theory has two parts: (1) a logic programming part which is able to express preferences and (2) a joint probability distribution. Probabilities are then used in order to determine the priorities of the possible alternatives.

6.3 P-log Modules

In this section, we define the notion of P-log modules and their semantics by means of a translation into logic program modules. Its probabilistic interpretation is provided by conditional probability measures. In what follows, we assume that different modules may have different sorted signatures Σ.

Definition 6.3.1 *A P-log module $\mathfrak{P}$ over Σ is a structure $\langle \Pi, Rin, Rout \rangle$ such that:*

1. *Π is a P-log program (possibly with observations and actions);*
2. *Rin is a set of ground attribute literals $a(\bar{t}) = y$, of random attributes declared in Π such that $y \in range(a)$;*
3. *$Rout$ is a set of ground Σ-literals, excluding attribute literals $a(\bar{t}) = y \in Rin$;*
4. *Π does not contain rules for any attribute $a(\bar{t})$ occurring in attribute literals of Rin, i.e., no random selection rule for $a(\bar{t})$, no regular rule with head $a(\bar{t}) = y$ nor a pr-atom for $a(\bar{t}) = y$.*

▲

The notion of P-log module is quite intuitive. First, the P-log program specifies the possible models and corresponding probabilistic information as before. However, the P-log module is parametric on a set of attribute terms *Rin*, which can be understood as the module's parent random variables. *Rout* specifies the random attributes which are visible as well as other derived logical conclusions. The last condition ensures that there is no interference between input and output random attributes.

The semantics of a P-log module is defined again in two stages. The possible worlds of a P-log module are obtained from the Answer Sets of a corresponding logic programming module. For simplifying definitions we assume that the isomorphism of attribute literals $a(\bar{t}) = y$ with $a(\bar{t}, y)$ instances is implicitly applied when moving from the P-log side to ASP side, and vice-versa.

Definition 6.3.2 *Consider a P-log module $\mathfrak{P} = \langle \Pi, Rin, Rout \rangle$ over signature Σ, and let $\mathbb{P}(\mathfrak{P}) = \langle R_\mathfrak{P}, I_\mathfrak{P}, O_\mathfrak{P}, H_\mathfrak{P} \rangle$ be the corresponding ASP module such that:*

1. *$R_\mathfrak{P}$ is $\tau(\Pi) \cup \{\leftarrow a(\bar{t}, y_1), a(\bar{t}, y_2) \mid a(\bar{t}) = y_1 \text{ and } a(\bar{t}) = y_2 \text{ in } Rin \text{ s.t. } y_1 \neq y_2\} \cup \{\leftarrow not\ hasval_\mathfrak{P}(a(\bar{t}))\} \cup \{hasval_\mathfrak{P}(a(\bar{t})) \leftarrow a(\bar{t}, y) \mid a(\bar{t}) = y \in Rin\}$, where predicates defining sorts have been renamed apart;*
2. *The set of input atoms $I_\mathfrak{P}$ of $\mathbb{P}(\mathfrak{P})$ is Rin.*
3. *The set of output atoms $O_\mathfrak{P}$ of $\mathbb{P}(\mathfrak{P})$ is Rout union all instances of $pa/3$ and $poss/3$ predicates of random attributes in Rout;*
4. *The set of hidden atoms $H_\mathfrak{P}$ of $\mathbb{P}(\mathfrak{P})$ is formed by $hasval_\mathfrak{P}/1$ instances for attribute literals in Rin, the Σ-literals not included in the output or input atoms of $\mathbb{P}(\mathfrak{P})$, with all sort predicates renamed apart.*

The possible models of $\mathfrak{P}$ are $\Omega(\mathfrak{P}) = \{M \cap (Rin \cup Rout) \mid M \in AS(\mathbb{P}(\mathfrak{P}))\}$. The name $hasval_\mathfrak{P}$ is local to $\mathfrak{P}$ and it does not occur elsewhere. ▲

The necessity of having sort predicates renamed apart is essential to avoid name clashes between different modules using the same sort attributes. Equivalently, the program can be instantiated, and all sort predicates removed. The extra integrity constraints in $R_\mathfrak{P}$ discard models where a random attribute has not exactly one assigned value. The set of input atoms in $\mathfrak{P}$ is formed by the random attribute literals in *Rin*. The set of output atoms includes all the instances of $pa/3$ and $poss/3$ in order to be possible to determine the causal probabilities in each model. By convention, all the remaining literals are hidden. A significant difference to the ordinary MLP modules is that the set of possible models are projected with respect to the visible literals, discarding hidden information in the models. The semantics of a P-log module is defined by probabilistic conditional measures:

Definition 6.3.3 *Consider a P-log module $\mathfrak{P} = \langle \Pi, Rin, Rout \rangle$ over signature Σ. Let E be any subset of $Rin \cup Rout$, and W be a possible world of P-log module $\mathfrak{P}$. If $E \subseteq W$ then the conditional unnormalised probability of W given E induced by $\mathfrak{P}$ is*

$$\hat{\mu}_\mathfrak{P}(W|E) = \sum_{M_i \in AS(\mathbb{P}(\mathfrak{P})) \text{ s.t. } M_i \cap (Rin \cup Rout) = W} \prod_{a(\bar{t}, y) \in M_i} \boldsymbol{P}(M_i, a(\bar{t}) = y)$$

where the product is taken over atoms for which $\boldsymbol{P}(M_i, a(\bar{t}) = y)$ is defined in M_i. Otherwise, $E \not\subseteq W$ and we set $\hat{\mu}_\mathfrak{P}(W|E) = 0.0$.

If there is at least one possible world with nonzero unnormalised conditional probability, for a particular E, then the conditional probability measure $\mu_\mathfrak{P}(.|E)$ is determined, and $\mu_\mathfrak{P}(W|E)$ for a possible world W given E induced by $\mathfrak{P}$ is:

$$\mu_\mathfrak{P}(W|E) = \frac{\hat{\mu}_\mathfrak{P}(W|E)}{\sum_{W_i \in \Omega(\mathfrak{P})} \hat{\mu}_\mathfrak{P}(W_i|E)} = \frac{\hat{\mu}_\mathfrak{P}(W|E)}{\sum_{W_i \in \Omega(\mathfrak{P}) \wedge E \subseteq W_i} \hat{\mu}_\mathfrak{P}(W_i|E)}.$$

When the P-log module $\mathfrak{P}$ is clear from the context we may simply write $\hat{\mu}(W|E)$ and $\mu(W|E)$ instead of $\hat{\mu}_\mathfrak{P}(W|E)$ and $\mu_\mathfrak{P}(W|E)$ respectively. ▲

An important remark regarding the above definition is that a possible world W of the P-log module can correspond to several models (the answer sets M_i) of the underlying answer set program, since hidden atoms have been projected out. This way, we need to sum the associated unconditional measures of the ASP models which originate (or contribute to) W. The attentive reader should have noticed that for any world W the unconditional probability measure $\hat{\mu}_{\mathfrak{P}}(W|E)$ and any $E \subseteq W \cap (Rin \cup Rout)$, is identical to $\hat{\mu}_{\mathfrak{P}}(W|W \cap (Rin \cup Rout))$, and zero elsewhere. So, in practice each world just requires one real value in order to obtain all conditional probability measures.

Example 6.3.1 *Construct P-log module* $\mathfrak{Sprinkler}$ *from Example 6.2.1 whose input atoms are* $\{cloudy = t, cloudy = f\}$ *and output atoms are* $\{sprinkler = t, sprinkler = f\}$. *The P-log program of* $\mathfrak{Sprinkler}$ *(with the observation removed) is*

$$
\begin{array}{l}
Boolean = \{t, f\}. \\
cloudy : Boolean. \\
sprinkler : Boolean. \\
[rs]\, random(sprinkler, \{X : Boolean(X)\}). \\
pr_{rs}(sprinkler = t \mid_c cloudy = f) = 0.5. \\
pr_{rs}(sprinkler = t \mid_c cloudy = t) = 0.1.
\end{array}
$$

For which, the corresponding ASP program in module $\mathbb{P}(\mathfrak{Sprinkler})$ *is:*

$$
\begin{array}{l}
hasval(cloudy) \leftarrow cloudy(t). \\
hasval(cloudy) \leftarrow cloudy(f).
\end{array}
$$

$$
\begin{array}{l}
\leftarrow not\ hasval(cloudy). \\
\leftarrow cloudy(t),\ cloudy(f).
\end{array}
$$

$$
\neg sprinkler(Y1) \leftarrow sprinkler(Y2),\ Y1 != Y2,\ Boolean(Y1),\ Boolean(Y2).
$$

$$
\begin{array}{l}
1\{sprinkler(Z) : poss(rsk, sprinkler, Z)\}1 \leftarrow not\ intervene(sprinkler). \\
poss(rsk, sprinkler, Z) \leftarrow Boolean(Z), not\ intervene(sprinkler). \\
intervene(sprinkler) \leftarrow do(sprinkler(Y)), Boolean(Y).
\end{array}
$$

$$
\begin{array}{l}
pa(rsk, sprinkler(t), 0.1) \leftarrow poss(rsk, sprinkler, t), cloudy(t). \\
pa(rsk, sprinkler(t), 0.5) \leftarrow poss(rsk, sprinkler, t), cloudy(f).
\end{array}
$$

$$
\begin{array}{l}
\leftarrow obs(sprinkler(t)), not\ sprinkler(t). \\
\leftarrow obs(sprinkler(f)), not\ sprinkler(f).
\end{array}
$$

Module $\mathbb{P}(\mathfrak{Sprinkler})$ *has four answer sets all containing both* $poss(rsk, sprinkler, t)$

and $poss(rsk, sprinkler, f)$*, and additionally:*

$$M_1 = \{sprinkler(t),\ cloudy(t),\ pa(rsk, sprinkler(t), 0.1)\}$$
$$M_2 = \{sprinkler(f),\ cloudy(t),\ pa(rsk, sprinkler(t), 0.1)\}$$
$$M_3 = \{sprinkler(t),\ cloudy(f),\ pa(rsk, sprinkler(t), 0.5)\}$$
$$M_4 = \{sprinkler(f),\ cloudy(f),\ pa(rsk, sprinkler(t), 0.5)\}$$

The first two correspond to possible worlds $W_1 = \{sprinkler(t),\ cloudy(t)\}$ *and* $W_2 = \{sprinkler(f),\ cloudy(t)\}$ *where* $cloudy = t$*. So,* $\hat{\mu}(W_1|\{cloudy(t)\}) = 0.1$ *and* $\hat{\mu}(W_2|\{cloudy(t)\}) = 0.9$ *and* $\hat{\mu}(W_3|\{cloudy(t)\}) = \hat{\mu}(W_4|\{cloudy(t)\}) = 0.0$*. Since the sum of the unconditional probability measures for all world totals 1.0, then the normalised measure coincides with the unnormalised one for the particular evidence* $\{cloudy = t\}$*.* △

Definition 6.3.4 (Conditional Probability) *Suppose* $\mathfrak{P}$ *is a P-log module and* $E \subseteq Rin \cup Rout$ *for which* $\mu_\Pi(.|E)$ *is determined. The probability,* $P_\mathfrak{P}(A|E)$*, of a formula* A *over literals in Rout, is the sum of the conditional probability measures of the possible worlds of* $\mathfrak{P}$ *on which* A *is true, i.e.,*

$$P_\mathfrak{P}(A|E) = \sum_{W \vdash A} \mu_\mathfrak{P}(W|E).$$

▲

The following theorem shows that P-log modules generalise appropriately the notion of conditional probability of P-log programs.

Theorem 6.3.1 *Let* Π *be P-log program* Π*. Consider the P-log module* $\mathfrak{P} = \langle \Pi, \{\}, Lit(\Sigma)\rangle$ *then for any set* $B \subseteq Lit(\Sigma)$ *such that* $P_\Pi(B) \neq 0$ *then* $P_\Pi(A|B) = P_\mathfrak{P}(A|B)$*.* ○

Proof of Theorem 6.3.1. By definition we have $P_\Pi(A|B) = P_\Pi(A \wedge B)/P_\Pi(B)$ whenever $P_\Pi(B) \neq 0$. Since the P-log module $\mathfrak{P}$ transforms all literals in the signature into output atoms, then the corresponding ASP module has no hidden and no input atoms. Thus the answer sets of $\mathbb{P}(\mathfrak{P})$ are exactly the possible worlds of P-log program $R_\mathfrak{P}$. Since there are no input atoms in the P-log module $\mathfrak{P}$ then $R_\mathfrak{P} = \tau(\Pi)$. Moreover, since there are no hidden atoms, then the possible worlds of $\mathfrak{P}$ are exactly the answer sets of $R_\mathfrak{P}$ i.e., the possible models of Π.

Since possible worlds of the P-log module are in one-to-one correspondence with the possible worlds of Π then $\hat{\mu}_\mathfrak{P}(W|B) = \hat{\mu}_\Pi(W)$, whenever $B \subseteq W$. But the worlds W for which $B \subseteq W$ are exactly the worlds where the conjunction of elements in B is true. Thus,

$$\mu_\mathfrak{P}(W|B) = \frac{\hat{\mu}_\mathfrak{P}(W|B)}{\sum_{W_i \in \Omega(\mathfrak{P}) \wedge B \subseteq W_i} \hat{\mu}_\mathfrak{P}(W_i|B)} = \frac{\hat{\mu}_\Pi(W)}{\sum_{W_i \vdash B} \hat{\mu}_\Pi(W_i)}$$

Therefore:

$$P_{\mathfrak{P}}(A|B) = \sum_{W \vdash A} \mu_{\mathfrak{P}}(W|B) = \frac{\sum_{W \vdash A \wedge B} \hat{\mu}_{\Pi}(W)}{\sum_{W_i \vdash B} \hat{\mu}_{\Pi}(W_i)} =$$

$$= \frac{\frac{\sum_{W \vdash A \wedge B} \hat{\mu}_{\Pi}(W)}{\sum_{W_i} \hat{\mu}_{\Pi}(W_i)}}{\frac{\sum_{W_i \vdash B} \hat{\mu}_{\Pi}(W_i)}{\sum_{W_i} \hat{\mu}_{\Pi}(W_i)}} = \frac{P_{\Pi}(A \wedge B)}{P_{\Pi}(B)}$$

□

A P-log module corresponds to the notion of factor introduced by Zhang & Poole (1996) in their variable elimination algorithm. The difference is that P-log modules are defined declaratively by a logic program with associated probabilistic semantics, instead of just matrixes of values for each possible combination of parameter variables.

6.4 P-log module theorem

This section provides a way of composing P-log modules and presents the corresponding module theorem. The composition of a P-log module mimics syntactically the composition of an answer set programming module, with similar pre-conditions:

Definition 6.4.1 (P-log module composition) *Consider P-log modules* $\mathfrak{P}_1 = \langle \Pi_1, Rin_1, Rout_1 \rangle$ *over signature* Σ_1*, and* $\mathfrak{P}_2 = \langle \Pi_2, Rin_2, Rout_2 \rangle$ *over signature* Σ_2*, such that:*

1. $Rout_1 \cap Rout_2 = \emptyset$
2. $(Lit(\Sigma_1) \setminus (Rin_1 \cup Rout_1)) \cap Lit(\Sigma_2) = Lit(\Sigma_1) \cap (Lit(\Sigma_2) \setminus (Rin_2 \cup Rout_2)) = \emptyset$
3. *The sorts of* Σ_1 *and* Σ_2 *coincide and are defined equivalently in* Π_1 *and* Π_2*.*

The composition of $\mathfrak{P}_1$ *with* $\mathfrak{P}_2$ *is the P-log module* $\mathfrak{P}_1 \oplus \mathfrak{P}_2 = \langle \Pi_1 \cup \Pi_2, (Rin_1 \cup Rin_2) \setminus (Rout_1 \cup Rout_2), (Rout_1 \cup Rout_2) \rangle$ *over signature* $\Sigma_1 \cup \Sigma_2$*.*

The join $\mathfrak{P}_1 \sqcup \mathfrak{P}_2 = \mathfrak{P}_1 \oplus \mathfrak{P}_2$ *is defined in this case whenever additionally there are no dependencies (positive or negative) among modules.* ▲

The first condition forbids the composition of modules having a common output literal, while the second one forbids common hidden atoms (except possibly the sort predicate instances). We lifted the first unnatural condition before but avoid here joining modules having both negative and positive dependencies.

The compositionality result for P-log modules is more intricate since besides the compositional construction of possible worlds, it is also necessary to ensure compositionality for the underlying conditional probability measures induced by the joined module:

Theorem 6.4.1 (P-log Module Theorem) *Consider two P-log modules $\mathfrak{P}_1$ and $\mathfrak{P}_2$ such that their join $\mathfrak{P}_1 \sqcup \mathfrak{P}_2$ is defined. Then, the possible worlds of their join is the natural join of their possible worlds:*

$$\Omega(\mathfrak{P}_1 \sqcup \mathfrak{P}_2) = \Omega(\mathfrak{P}_1) \bowtie \Omega(\mathfrak{P}_2)$$

Where:

$$\Omega(\mathfrak{P}_1) \bowtie \Omega(\mathfrak{P}_2) = \{W_1 \cup W_2 \mid W_1 \in \Omega(\mathfrak{P}_1), W_2 \in \Omega(\mathfrak{P}_1)$$

and:

$$W_1 \cap (Rin_2 \cup Rout_2) = W_2 \cap (Rin_1 \cup Rout_1)\}$$

Let now $E = E_1 \cup E_2$ where: $E_1 = E \cap (Rin_1 \cup Rout_1)$ and $E_2 = E \cap (Rin_2 \cup Rout_2)$ Then:

$$\hat{\mu}_{\mathfrak{P}_1 \sqcup \mathfrak{P}_2}(W|E) = \hat{\mu}_{\mathfrak{P}_1}(W_1|E_1) \times \hat{\mu}_{\mathfrak{P}_2}(W_2|E_2) \text{ with } W = W_1 \cup W_2$$

such that:

$$W \in \Omega(\mathfrak{P}_1 \sqcup \mathfrak{P}_2,\ W_1 \in \Omega(\mathfrak{P}_1) \text{ and } W_2 \in \Omega(\mathfrak{P}_2)$$

∘

Proof of Theorem 6.4.1. Under the conditions of the theorem, the join of ASP modules $\mathbb{P}(\mathfrak{P}_1)$ and $\mathbb{P}(\mathfrak{P}_2)$ is defined, and the answer sets of $AS(\mathbb{P}(\mathfrak{P}_1) \sqcup \mathbb{P}(\mathfrak{P}_2)) = AS(\mathbb{P}(\mathfrak{P}_1)) \bowtie AS(\mathbb{P}(\mathfrak{P}_2))$. However the answer sets of $R_{\mathfrak{P}_1 \sqcup \mathfrak{P}_2}$ are the answer sets of $R_{\mathfrak{P}_1} \cup R_{\mathfrak{P}_2}$ because the only difference between the programs is possibly the integrity constraints imposed to an input attribute literal in $R_{\mathfrak{P}_1}$ (or $R_{\mathfrak{P}_2}$) which might be absent from $R_{\mathfrak{P}_1} \cup R_{\mathfrak{P}_2}$ because the input attribute literal is an output literal in the other module. However, if it is an output attribute literal there must exist a random selection rule for it which imposes at least the constraint in the other program; so the claim holds.

Let $M \in AS(\mathbb{P}(\mathfrak{P}_1) \sqcup \mathbb{P}(\mathfrak{P}_2))$ then by the ASP module theorem we know that $M = M_1 \cup M_2$ such that $M_1 \in AS(\mathbb{P}(\mathfrak{P}_1))$ and $M_2 \in AS(\mathbb{P}(\mathfrak{P}_2))$ such that M_1 is compatible with M_2 (i.e., coincide in common atoms). If we eliminate hidden atoms of W we obtain a possible world W of $\mathbb{P}(\mathfrak{P}_1) \sqcup \mathbb{P}(\mathfrak{P}_2)$, i.e., $W = M \cap (Rin_1 \cup Rout_1 \cup Rin_2 \cup Rout_2)$. Moreover, $W_1 = M_1 \cap (Rin_1 \cup Rout_1 \cup Rin_2 \cup Rout_2) = M_1 \cap (Rin_1 \cup Rout_1)$ is a possible world of $\mathfrak{P}_1$, and similarly $W_2 = M_2 \cap (Rin_2 \cup Rout_2)$ is a possible world of $\mathfrak{P}_2$, otherwise there would be a hidden literal in one module belonging to the literals of the other module. Therefore:

$$\begin{aligned} W &= M \cap (Rin_1 \cup Rout_1 \cup Rin_2 \cup Rout_2) \\ &= (M_1 \cup M_2) \cap (Rin_1 \cup Rout_1 \cup Rin_2 \cup Rout_2) \\ &= (M_1 \cap (Rin_1 \cup Rout_1 \cup Rin_2 \cup Rout_2)) \cup \\ &\quad (M_2 \cap (Rin_1 \cup Rout_1 \cup Rin_2 \cup Rout_2)) \\ &= (M_1 \cap (Rin_1 \cup Rout_1)) \cup (M_2 \cap (Rin_2 \cup Rout_2)) = W_1 \cup W_2 \end{aligned}$$

By the conditions imposed to the join of P-log modules it is clear that no common atom to M_1 and M_2 will be eliminated from M when projecting wrt to the visible

literals of the join. Therefore W_1 is compatible with W_2 and $W = W_1 \cup W_2$ regarding compositionality of unnormalised conditional probability measures. □

Notice that the P-log module theorem is defined only in terms of the unnormalised conditional probability measures. The normalised ones can be obtained as in the previous case dividing by the sum of unconditional measure of all worlds given the evidence. Again, we just have to consider one value for each world (i.e., when evidence is maximal).

The application to Bayesian Networks is now straightforward. First, each random variable in a Bayesian Network is captured by a P-log module having the corresponding attribute literals of the random variable as output literals, and input literals are all attribute literals obtainable from parent variables. The conditional probability tables are represented by pr-atoms, as illustrated before in Example 6.2.1. P-log module composition inherits associativity and commutativity from ASP modules, and thus P-log modules can be joined in arbitrary ways since there are no common output atoms, and there are no cyclic dependencies.

The important remark is that a P-log module is an extension of the notion of factor used in the variable elimination algorithm Zhang & Poole (1996). We only need a way to eliminate variables from a P-log module in order to simulate the behaviour of the variable elimination algorithm, but this is almost trivial:

Definition 6.4.2 (Elimination Operation) *Consider a P-log module $\mathfrak{P} = \langle \Pi, Rin, Rout \rangle$ over signature Σ, and a subset of attribute literals $S \subseteq Rout$. Then, P-log module $Elim(\mathfrak{P}, S) = \langle \Pi, Rin, Rout \setminus S \rangle$ eliminates (hides) from $\mathfrak{P}$ the attribute literals in S.* ▲

By hiding all attribute literals of a given random variable, we remove the random attribute from the possible worlds (as expected), summing away corresponding original possible worlds. By applying the composition of P-log modules and eliminate operations by the order they are performed by the variable elimination algorithm, the exact behavior of the variable elimination algorithm is attained.

Example 6.4.1 (Variable Elimination) *Take the following joint probability distribution. A variable, v can be summed out between a set of instantiations where the set $V \setminus v$ must agree over the remaining variables. The value of variable v becomes irrelevant if it is the variable to be summed out.*

V_1	V_2	V_3	V_4	V_5	$Pr(.)$
$true$	$true$	$true$	$false$	$false$	0.80
$false$	$true$	$true$	$false$	$false$	0.20

After eliminating V_1, its reference is excluded and we are left with a distribution only over the remaining variables and the sum of each instantiation.

V_2	V_3	V_4	V_5	$Pr(.)$
$true$	$true$	$false$	$false$	1.0

The resulting distribution which follows the sum-out operation only helps to answer queries that do not mention V_1. Also worthy to note, the summing-out operation is commutative. △

Thus, for the case of Bayesian Networks with a polytree structure where each variable corresponds to a P-log module, and all the conditional probability tables are encoded with explicit pr-atoms, reasoning can be performed in polynomial time on the size of the modules by eliminating at each stage a singly connected variable. Reasoning will be exponential in the maximum number of parents of a variable, i.e. input atoms, but since the full CPTs is encoded in the program the polynomial complexity result follows.

6.5 Conclusions and Future Work

We present the first approach in the literature to modularise P-log programs and to make their composition incrementally by combining compatible possible worlds and multiplying corresponding unnormalised conditional probability measures. A P-log module corresponds to a factor of the variable elimination algorithm Zhang & Poole (1996); Poole & Zhang (2011), clarifying and improving the relationship of P-log with traditional Bayesian Network approaches. By eliminating variables in P-log modules we may reduce the space and time necessary to make inference in P-log, in contrast with previous algorithms Anh *et al.* (2008); Gelfond *et al.* (2006) which require always enumeration of the full possible worlds (which are exponential on the number of random variables) and repeat calculations. As expected, it turns out that the general case of exact inference is intractable, so we must consider methods for approximate inference.

6.5.1 Future Work

We intend to fully describe the inference algorithm obtained from the compositional semantics of P-log modules and relate it formally with the variable elimination algorithm. Furthermore we expect that the notion of P-log module may also help to devise approximate inference methods, e.g., by extending sampling algorithms, enlarging the applicability of P-log which is currently somehow restricted. We hope to generalise the P-log language to consider other forms of uncertainty representation like belief functions, possibility measures or even plausibility measures Fagin & Halpern (1994).

The ways with which we deal with conflicts that arise with both positive and negative cycles, in the following Part III, can potentially allow us to avoid the restrictions we impose on modular P-log programs, forcing them to capture Bayesian Networks and thus have no cycles. This direction may lead us to a way of dealing with hidden Markov models (HMM) which are statistical Markov models in which the system being modeled is assumed to be a Markov process with unobserved (hidden) states.

HMMs can be considered as the simplest dynamic Bayesian networks and are especially known for their application in temporal pattern recognition such as speech, handwriting, gesture recognition, part-of-speech tagging, musical score following, partial discharges and bioinformatics.

Still as future work we can straightforwardly extend these results to probabilistic reasoning with ASP by applying the new module theorem to the work we presented in Chapters 4 and 5, as well as to DLP functions and general answer sets. An implementation of the framework is also foreseen in order to assess the overhead when compared with the original benchmarks in Oikarinen & Janhunen (2008).

In the near future we plan to use the PRISM ideas to expand the semantics of P-log to allow infinite possible worlds. Our more distant plans include investigation of possible adaptation of PRISM statistical learning algorithms to P-log.

Part III

Conflicts in Answer Set Programming

Part Overview:

When combining LPs from different origins, conflicts may occur. There are, in general, two ways to solve them which we study in this Part III:

(1) Syntactically: By applying changes to the program modules, providing justifications and debugging models.

(2) Semantically: By using paraconsistent/paracoherent semantics.

The Barber Paradox

> "The barber is a man in town who shaves all those, and only those, men in town who do not shave themselves. Who shaves the barber?"

Example 6.5.1 *Consider the following logic program capturing this paradox:*

$$P = \left\{ \begin{array}{rl} shaves(barber,X) & \leftarrow male(X), not\ shaves(X,X). \\ shaved(X) & \leftarrow shaves(Y,X). \\ unshaved(X) & \leftarrow male(X), not\ shaved(X). \\ male(barber). & \end{array} \right\}$$

△

The program has no models due to incoherence and as we said before, this can be dealt with in two different ways:

(1) Syntactically: By applying changes to the program either by removing rules or by adding facts.

Example 6.5.2 *Say that our intuitions asks for the barber to be* Shaved. *We can simply add a module containing fact* shaves(barber,barber) *to our modular system, but that goes against the statement that says that "the barber only shaves those who do not shave themselves". It is thus a fix, yes, but one which implies going against what we wanted to represent in the first place i.e.,* $AS(P \cup \{shaves(barber,barber)\}) =$

$$\{\{male(barber), shaves(barber,barber), shaved(barber)\}\}$$

△

Example 6.5.3 *Say that our intuitions asks for the barber to be* Unshaved. *Removing the first or second rules, or fact* $male(barber)$*, from the program (thus obtaining program* P'*) are the most intuitive actions.*

$$AS(P') = \{\{male(barber), unshaved(barber)\}\}$$

△

(2) Semantically: By using a paracoherent semantics to evaluate the original program.

Example 6.5.4 *Semi-Stable Model semantics does not propagate belief (in the formed of K atoms) and provides a single model to this program:*

$$\{male(barber), Kshaves(barber, barber), unshaved(barber)\}$$

In this case, $Kshaves(barber, barber)$ does not provide support to $shaved(barber)$ and so, we can conclude that the barber is unshaved even though we believe that he shaves himself. △

Example 6.5.5 *A better alternative is using a semantic that propagates belief. Our Semi-Equilibrium Model semantics also provides one module:*

$$\{male(barber), Kshaves(barber, barber), Kshaved(barber)\}$$

In this case, $Kshaves(barber, barber)$ provides support to $Kshaved(barber)$ and so, we can conclude that we believe the barber is shaved because of our belief that he shaves himself. △

Outline We start by studying paracoherent semantics for ASP in Chapter 7. These are semantics that ascribes models to (disjunctive) logic programs with non-monotonic negation even if no answer set exists, due to a lack of stability in models caused by cyclic dependency through negation, or due to constraints. Ideally, such semantics approximates the answer set semantics faithfully and delivers models whenever possible; this can be beneficially exploited in scenarios where unexpected inconsistency arises and one needs to stay operational. Among few well-known semantics which feature these properties are the semi-stable model semantics Sakama & Inoue (1995a), and our novel semi-equilibrium model semantics, which amends the semi-stable model semantics by eliminating some anomalies. For both semantics, which are defined by program transformations, we present model-theoretic characterisations.

In Chapter 8 we provide a transformation to compute why not provenance models under the well-founded and the answer set semantics by computing the answer sets of meta-programs that capture the original programs and include some necessary extra atoms. We do this in a modular way, preserving compatibility with the previous work of Viegas Damásio *et al.* (2013) and computing the models directly without first obtaining provenance formulas for interpretations which enables computing provenance answer sets in an easy way by using AS solvers. Having this, we then align provenance and debugging answer sets in a unified transformation and show that the provenance approach generalises the debugging one, since any error has a counterpart provenance but not the other way around. Our mapping allows generating answer sets capturing errors and justifications for (intended) models. As expected, they are exponential.

Then, in Chapter 9, we identify different types of basic conflicts that occur in a real-world scenario of access control programs and characterise them in terms of a

(relativised) notion of *strong equivalence* of logic programs Lifschitz *et al.* (2000); Eiter *et al.* (2007). We will also identify conflicts that occur when we introduce default negation and characterise them in terms of default logic. These characterisations potentially enable the detection of conflicts to be done automatically by using automatic theorem provers. Overall, these characterisations are flexible enough to be extended to several types of conflicts and can be used to detect which types of conflict are generated, as well as trace them back to the sources. This can be done by linking these characterisations with our unified justifications and debugging framework and is left for future work.

Chapter 7

Paracoherent Answer Set Programming

The answer set semantics may assign a logic program no model, due to logical contradiction or unstable negation, which is caused by cyclic dependency of an atom from its negation. While logical contradictions can be handled with traditional techniques from paraconsistent reasoning, instability requires other methods. We consider resorting to a paracoherent semantics, in which 3-valued interpretations are used where a third truth value besides true and false expresses that an atom is believed true. This is at the basis of the semi-stable model semantics, which was defined using a program transformation. In this chapter, we give a model-theoretic characterisation of semi-stable models, which makes the semantics more accessible. Motivated by some anomalies of semi-stable model semantics with respect to basic epistemic properties, we propose an amendment that satisfies these properties. The latter has both a transformational and a model-theoretic characterisation that reveals it as a relaxation of equilibrium logic, the logical reconstruction of answer set semantics, and is thus called the semi-equilibrium model semantics. A complexity analysis of major reasoning tasks shows that semi-equilibrium models are harder than answer sets (i.e., equilibrium models), due to a global minimisation step for keeping the gap between true and believed true atoms as small as possible. Our results contribute to the logical foundations of paracoherent answer set programming, which gains increasing importance in inconsistency management, and at the same time provide a basis for algorithm development and integration into answer set solvers.

7.1 Introduction

Answer Set Programming (ASP) is a declarative programming paradigm with a model-theoretic semantics, where problems are encoded using rules, and its models encode solutions. However, due to non-monotonicity, programs may be incoherent, i.e., lack an answer set due to cyclic dependencies of an atom from its default nega-

tion. Nonetheless, there are many cases when this is not intended and one might want to draw conclusions also from an incoherent program, e.g., for debugging purposes, or in order to keep a system (partially) responsive in exceptional situations. This is akin to the principle of paraconsistency, where non-trivial consequences shall be derivable from an inconsistent theory. As so-called extended logic programs also may be inconsistent in the classical sense, i.e., they may have the inconsistent answer set as their unique answer set, we use the term *paracoherent reasoning* to distinguish between paraconsistent reasoning and reasoning from incoherent programs.

Both types of reasoning from answer set programs have been studied in the course of the development of the answer set semantics; for approaches on paraconsistent ASP cf., e.g., Sakama & Inoue (1995a), Alcântara *et al.* (2005), Odintsov & Pearce (2005)). Numerous semantics for logic programs with nonmonotonic negation can be considered as a paracoherent semantics for ASP. Ideally, such a semantics satisfies the following properties:

1. Every (consistent) answer set of a program corresponds to a model (*answer set coverage*).

2. If a (consistent) answer set exists for a program, then all models correspond to an answer set (*congruence*).

3. If a program has a classical model, then it has a model (*classical coherence*).

Widely-known semantics, such as 3-valued stable models Przymusinski (1991a), L-stable models Eiter *et al.* (1997b), revised stable models Pereira & Pinto (2005), regular models You & Yuan (1994), and pstable models Osorio *et al.* (2008), satisfy only part of these requirements (see the Discussion section for further semantics and more details). Semi-stable models Sakama & Inoue (1995a) however, satisfy all three properties and thus are the prevailing paracoherent semantics.

Despite the model-theoretic nature of ASP, semi-stable models have been defined by means of a program transformation, called epistemic transformation. A semantic characterisation in the style of equilibrium models for answer sets Pearce & Valverde (2008) is still missing. In this chapter, we address this problem and present the following main contributions.

– We characterise semi-stable models by pairs of 2-valued interpretations of the original program, similar to so-called here-and-there (HT) models. In that, we point out some anomalies of the semi-stable semantics with respect to basic rationality properties in modal logics (**K** and **N**), that essentially prohibit a 1-to-1 characterisation in terms of HT-models.

– This leads us to propose an alternative paracoherent semantics, called semi-equilibrium semantics, which satisfies the aforementioned properties and can be characterised using HT-models. Moreover, semi-equilibrium models can be obtained by selecting answer sets of an extension of the epistemic transformation (applying the same criteria as for semi-stable models).

Our results contribute to a more logical foundation of paracoherent answer set programming, which gains increasing importance in inconsistency management.

7.1.1 Use case scenarios

Paracoherent semantics may be fruitfully employed in different use cases of ASP, such as model building respectively scenario generation, but also traditional reasoning from the models of a logical theory. The semi-stable model semantics is attractive as it

(1) brings in "unsupported" assumptions as being believed,

(2) remains close to answer sets in model building, but distinguishes atoms that require such assumptions from atoms derivable without them, not creating justified truth from positive beliefs, and

(3) keeps the closed world assumption[1] (CWA)/LP spirit of minimal assumptions.

Let us now consider the following two scenarios.

7.1.1.1 Model building

In ASP, one of the principal reasoning tasks is model building, which means to compute some, multiple or even all answer sets of a given program. Each answer set encodes a possible world or solution to a problem that is represented by the program.

The standard answer set semantics may be regarded as appropriate when a knowledge base, i.e., logic program, is properly specified adopting the CWA principle to deal with incomplete information. It may then be perfectly ok that no answer set exists, as e.g. in the *theoretical exercise* of the barber paradox. However, sometimes the absence of an answer set is unacceptable as a possible world is known to exist, and in this case a relaxation of the answer set semantics is desired.

Example 7.1.1 *Suppose we have a program that captures knowledge about friends of a person regarding visits to a party, where $go(X)$ informally means that X will go:*

$$P = \left\{ \begin{array}{c} go(John) \leftarrow not\ go(Mark). \\ go(Peter) \leftarrow go(John), not\ go(Bill). \\ go(Bill) \leftarrow go(Peter). \end{array} \right\}$$

It happens that P has no answer set. This is unacceptable as we know that there is a model in reality, regardless of who will go to the party*, and we need to cope with this situation. Semi-stable semantics is a tool that allows us to gain an answer set, by relaxing the CWA and adopting beliefs without further justifications. In particular, the semi-stable models of this program are $I_1^K = \{Kgo(Mark)\}$ and $I_2^K = \{go(John), Kgo(John), Kgo(Bill)\}$. Informally, the key difference between I_1^K and I_2^K concerns the beliefs on Mark and John. In I_2^K Mark does not go, and, consequently, John will go (moreover, Bill is believed to go, and Peter will not go). In I_1^K, instead, we believe Mark will go, thus John will not go (likewise Peter and Bill). Notably, and different*

[1] If every rule with an atom a in the head has a false body, or its head contains a true atom distinct from a with respect to an acceptable model, then a must be false in that model.

from other related formalisms (cf. Section 7.6.2), positive beliefs do not create justified truth: if we had a further rule fun ← go(Mark) in P, then from just believing that Mark will go we can not derive that fun is true; I_1^K would remain a semi-stable model. △

As already mentioned, paracoherent semantics can serve as a starting point for debugging and also repairing a program. Indeed, if all believed atoms were justified true, then we would obtain an answer set of the program.[2] Therefore, we might investigate reasons for the failure to derive these atoms justified, and possibly add new rules or modify existing ones. However, fully dealing with this issue and linking it to existing work on debugging and repair of answer set programs is beyond the scope of this thesis; we will briefly address it in Section 7.6.2).

7.1.1.2 Inconsistency-tolerant query answering

Query answering over a knowledge base resorts usually to brave or, respectively, to cautious inference from the answer sets of a knowledge base, where the query has to hold in some, respectively in every, answer set; let us focus on the latter here. However, if incoherence of the knowledge base arises, then we lose all information and query answers are trivial, since every query is vacuously false (respectively vacuously true). This, however, may not be satisfactory and be problematic, especially if one can not modify the knowledge base, which may be due to various reasons (no permission for change, the designer/administrator of the knowledge base might be unavailable etc). Paracoherent semantics can be exploited to overcome this problem and to render query answering operational, without trivialisation. We illustrate this on an extension to the barber paradox (but could equally well consider other scenarios).

Example 7.1.2 *Consider a variant of the barber paradox, cf. Sakama & Inoue (1995a):*

$$P = \left\{ \begin{array}{ll} shaves(joe,X) \leftarrow & not\ shaves(X,X),\ man(X). \\ man(paul). & \\ man(joe). & \end{array} \right\}$$

While this program has no answer set, the semi-stable model semantics gives us the model $\{man(joe),\ shaves(joe,paul),\ man(paul),\ Kshaves(joe,joe)\}$*, in which* $shaves(joe,joe)$ *is believed to be true (as expressed by the prefix 'K'); here the incoherent rule* $shaves(joe,joe) \leftarrow not\ shaves(joe,joe), man(joe)$*, which is an instance of the rule in P for joe, is isolated from the rest of the program to avoid the absence of models;*[3] *this treatment allows us to derive, for instance, that* $shaves(joe,paul)$ *and* $man(paul)$ *are true; furthermore, we can infer that* $shaves(joe,joe)$ *can not be false. Such a capability seems to be very attractive in query answering: to* tolerate inconsistency (that is, incoherence) without a "knowledge exploision." △

[2] As we shall see, this actually holds for the amended semi-stable semantics.

[3] A similar intuition underlies the CWA inhibition rule in Pereira *et al.* (1992) that is used for contradiction removal in logic programs.

The well-founded semantics (WFS) van Gelder *et al.* (1991), which is the most prominent approximation of the answer set semantics, has similar capabilities, but takes intuitively a coarser view on the truth value of an atom, which can be either true, false, or undefined; in semi-stable semantics, however, undefinedness has a bias towards truth, expressed by "believed true" (or stronger, by "must be true"); in the example above, under WFS $shaves(joe,joe)$ would be undefined. Furthermore, undefinedness is cautiously propagated, which may prevent one from drawing expected conclusions.

Example 7.1.3 *Consider the following extension of Russell's paraphrase:*

$$P = \left\{ \begin{array}{l} shaves(joe,joe) \leftarrow not\ shaves(joe,joe). \\ visits\_barber(joe) \leftarrow not\ shaves(joe,joe). \end{array} \right\}.$$

Arguably one expects that $visits\_barber(joe)$ is concluded false from this program: to satisfy the first rule, $shaves(joe,joe)$ can not be false, and thus the second rule can not be applied; thus under CWA, $visits\_barber(joe)$ should be false. However, under well-founded semantics all atoms are undefined; in particular, the undefinedness of $shaves(joe,joe)$ is propagated to $visits\_barber(joe)$ by the second rule.

The single semi-stable model of P from its epistemic transformation is $\{Kshaves(joe,joe)\}$, according to which $shaves(joe,joe)$ is believed true while $visits\_barber(joe)$ is false. △

Furthermore, it is well-known that the well-founded semantics has problems with reasoning by cases.

Example 7.1.4 *From the program*

$$P = \left\{ \begin{array}{l} shaves(joe,joe) \leftarrow not\ shaves(joe,joe); \\ angry(joe) \leftarrow not\ happy(joe).\ happy(joe) \leftarrow not\ angry(joe). \\ smokes(joe) \leftarrow angry(joe);\ smokes(joe) \leftarrow happy(joe). \end{array} \right\},$$

which is still incoherent with respect to answer set semantics, we can not conclude that $smokes(joe)$ is true under WFS, while we can do so under semi-stable semantics and its relatives. Moreover, under these semantics we can not derive that $smokes(joe)$ is true, which means that trivialisation is avoided. △

7.2 Semi-Stable Models

Sakama & Inoue (1995a) introduced *semi-stable models* as an extension of paraconsistent answer set semantics (called PAS semantics, respectively p-stable models by the authors) for extended disjunctive logic programs. Their aim was to provide a framework which is paraconsistent for incoherence, i.e., in situations where stability fails due to cyclic dependencies of a literal from its default negation.

Since we are primarily interested in paracoherence, in the following summary and study of semi-stable semantics, we disregard aspects devoted to paraconsistency, more

specifically, we exclude strong negation. Note also that Sakama & Inoue (1995a) allowed for programs with variables, while we focus on the propositional case. These restrictions help to put their technique to handle incoherence in perspective. Moreover, our results easily carry over to the original setting considering PAS semantics and allowing for non-ground programs. This will be considered in more detail in the Discussion section below.

We consider an extended propositional language $\Sigma^K = \Sigma \cup \{Ka \mid a \in \Sigma\}$. Intuitively, Ka can be read as a is believed to hold. Semantically, we resort to subsets of Σ^K as interpretations I^K and the truth values false $\bot$[4], believed true **bt**, and true **t**, where $\bot \preceq \mathbf{bt} \preceq \mathbf{t}$. The truth value assigned by I^K to a propositional variable a is defined by

$$I^K(a) = \begin{array}{l} \mathbf{t} \text{ if } a \in I^K, \\ \mathbf{bt} \text{ if } Ka \in I^K \text{ and } a \notin I^K, \\ \bot \text{ otherwise.} \end{array}$$

The semi-stable models of a program P are defined via its *epistemic transformation* P^K.

Definition 7.2.1 (P^K **Sakama & Inoue (1995a)**) *Let P be a disjunctive program. Then its epistemic transformation is defined as the positive disjunctive program P^K obtained from P by replacing each rule r of the form (2.2) in P, such that $Body^-(r) \neq \emptyset$, with:*

$$\lambda_{r,1} \vee \ldots \vee \lambda_{r,l} \vee Kb_{m+1} \vee \ldots \vee Kb_n \leftarrow b_1, \ldots, b_m, \tag{7.1}$$

$$a_i \leftarrow \lambda_{r,i}, \tag{7.2}$$

$$\leftarrow \lambda_{r,i}, b_j, \tag{7.3}$$

$$\lambda_{r,i} \leftarrow a_i, \lambda_{r,k}, \tag{7.4}$$

for $1 \leq i \leq l$, $m+1 \leq j \leq n$, and $1 \leq k \leq l$

▲

Intuitively, the atom Kc_j means that c_j must be believed to be true, and $\lambda_{r,i}$ means that in the rule r, some atom a_i in the head must be true. With this meaning, the rule (2.2) is naturally translated into the rule (7.1): if all atoms in $Body(r)$ are true, then either some atom in $Head(r)$ is true, and thus some $\lambda_{r,i}$ is true, or some atom c_i in $Body^-(r)$ must be believed to be true (then *not* c_i is false). The rule (7.2) propagates the value of $\lambda_{r,i}$ to a_i, which then is visible also in other rules. The other rules restrict the choice of λ_{r_i} for making the head of r true: if c_j is true, the rule r is inapplicable and no atom in $Head(r)$ has to be true (7.3). Furthermore, if the atom a_i in the head is true (via some other rule of P or by (7.2)), then whenever some atom a_k in $Head(r)$ must be true, then also a_i must be true (7.4); the minimality of answer set semantics effects that only a_i must be true.

[4] In Sakama & Inoue (1995a) $\bot$ is called 'undefined', as it should be if strong negation is considered as well.

Note that for any program P, its epistemic transformation P^K is positive. Models of P^K are defined in terms of a fixed point operator in Sakama & Inoue (1995a), with the property that for positive programs, according to Theorem 2.9, minimal fixed points coincide with minimal models of the program. Therefore, for any program P, minimal fixed points of P^K coincide with answer sets of P^K.

Semi-stable models are then defined as *maximal canonical* interpretations among the minimal fixed points (answer sets) of P^K as follows:

Definition 7.2.2 (Maximal Canonical) *Given an interpretation I^K over $\Sigma' \supseteq \Sigma^K$, let $gapI^K = \{Ka \mid Ka \in I^K \text{ and } a \notin I^K\}$. Given a set S of interpretations over Σ', an interpretation $I^K \in S$ is maximal canonical in S iff there is no interpretation $J^K \in S$ such that $gapI^K \supset gapJ^K$.* ▲

Let $mc(S)$ denote maximal canonical interpretations in S and let $\mathcal{SST}(P)$ be the semi-stable models of a program P, then we can equivalently paraphrase the definition of semi-stable models in Sakama & Inoue (1995a) as follows.

Definition 7.2.3 *Let P be a program over Σ. Then, $\mathcal{SST}(P) = \{I^K \cap \Sigma^K \mid I^K \in mc(\mathcal{AS}(P^K))\}$.* ▲

Example 7.2.1 *Reconsider $P = \{a \leftarrow not\ a\}$, where a stands for $shaves(joe, joe)$. Then $P^K = \{\lambda_1 \vee K_a \leftarrow .\ \ a \leftarrow \lambda_1.\ \ \leftarrow a, \lambda_1.\ \ \lambda_1 \leftarrow a, \lambda_1\}$, which has the single answer $M = \{Ka\}$; hence, $\{Ka\}$ is the single semi-stable model of P.* △

Example 7.2.2 *Consider the simple stratified program $P = \{b \leftarrow not\ a\}$. Its epistemic transformation is $P^K = \{\lambda_1 \vee Ka \leftarrow .\ \ b \leftarrow \lambda_1.\ \ \leftarrow a, \lambda_1.\ \ \lambda_1 \leftarrow b, \lambda_1.\}$, which has the answers sets $M_1 = \{Ka\}$ and $M_2 = \{\lambda_1, b\}$; as $gap(M_1) = \{a\}$ and $gap(M_2) = \emptyset$, among them M_2 is maximal canonical, and hence $M2 \cap \Sigma^K = \{b\}$ is the single semi-stable model of P. This is in fact also the unique answer set of P.* △

For a study of the semi-stable model semantics, we refer to Sakama & Inoue (1995b); notably,

Proposition 3 (Sakama & Inoue (1995b)) *The SST -models semantics, given by $SST(P)$ for arbitrary programs P, satisfies properties (D1)-(D3).* ○

7.3 Semantic Characterisation

As opposed to its transformational definition, in this work we aim at a model-theoretic characterisation of semi-stable models in the line of model-theoretic characterisations of the answer set semantics by means of HT.

Example 7.3.1 *Let $P = \{a \leftarrow not\ a\}$. The program is incoherent, with $\{Ka\}$ as its unique semi-stable model. Its HT-models are $(\emptyset, \{a\})$ and $(\{a\}, \{a\})$. One might aim characterising the semi-stable model by $(\emptyset, \{a\})$.* △

However, resorting to HT-interpretations will not uniquely characterise semi-stable models as illustrated next.

Example 7.3.2 *Consider now the following program*

$$P = \{a.\ \ b.\ \ c.\ \ d \leftarrow not\ a, not\ b.\ \ d \leftarrow not\ b, not\ c.\}.$$

It is coherent, with a single answer set $\{a,b,c\}$*, while its semi-stable models are:*

$$\mathcal{SST}(P) = \{\{a,b,c,Kb\},\{a,b,c,Ka,Kc\}\}$$

Note that neither $(\{a,b,c\},\{b\})$ *nor* $(\{a,b,c\},\{a,c\})$ *is a HT-interpretation.* △

Hence, for a 1-to-1 characterisation we have to resort to different structures. Sticking to the requirement that, given a program P over Σ, pairs of two-valued interpretations over Σ should serve as the underlying semantic structures, we say that a bi-interpetation of a program P over Σ is any pair (I,J) of interpretations over Σ, and define:

Definition 7.3.1 *Let* ϕ *be a formula over* Σ*, and let* (I,J) *be a bi-interpretation over* Σ*. Then,* (I,J) *is a* bi-model *of* ϕ*, denoted* $(I,J) \models_\beta \phi$ *iff*

1. $(I,J) \models_\beta a$ *if* $a \in I$*, for any atom* a*,*
2. $(I,J) \not\models_\beta \bot$*,*
3. $(I,J) \models_\beta \neg\phi$ *if* $J \not\models \phi$*,*
4. $(I,J) \models_\beta \phi \wedge \psi$ *if* $(I,J) \models_\beta \phi$ *and* $(I,J) \models_\beta \psi$*,*
5. $(I,J) \models_\beta \phi \vee \psi$ *if* $(I,J) \models_\beta \phi$ *or* $(I,J) \models_\beta \psi$*,*
6. $(I,J) \models_\beta \phi \rightarrow \psi$ *if:*
   - (*i*) $(I,J) \not\models_\beta \phi$*, or*
   - (*ii*) $(I,J) \models_\beta \psi$ *and* $I \models \phi$*.*

Moreover, (I,J) *is a bi-model of a program* P*, iff* $(I,J) \models_\beta \phi$*, for all* ϕ *of the form (2.5) corresponding to a rule* $r \in P$*.* ▲

Note that the only difference in the recursive definition of bi-models and HT-models is in item 6, i.e., the case of implication. While HT-models require that the material implication $\phi \rightarrow \psi$ holds in the *there*-world, *bi*-models miss such a connection between ϕ and ψ. This makes it possible that a bi-interpretation (I,J) such that $I \subseteq J$ is a bi-model but not an HT-model of an implication (2.5); a simple example is given by $(\{\},\{a\})$ and $a \rightarrow b$. On the other hand, each HT-model of an implication (2.5) is also a bi-model of it.

In case of programs, its bi-models can alternatively be characterised by the following condition on its rules.

Proposition 4 *Let r be a rule over Σ, and let (I,J) be a bi-interpretation over Σ. Then, $(I,J) \models_\beta r$ if and only if:*

$$(\mathbf{a})\ Body^+(r) \subseteq I \text{ and } J \cap Body^-(r) = \emptyset \text{ implies:}$$

$$I \cap Head(r) \neq \emptyset \text{ and } I \cap Body^-(r) = \emptyset$$

∘

Proof of Proposition 4. Let r be a rule over Σ, and let (I,J) be a bi-interpretation over Σ.

($\Leftarrow$) Suppose that (I,J) satisfies (a), i.e., $Body^+(r) \subseteq I$ and $J \cap Body^-(r) = \emptyset$ implies $I \cap Head(r) \neq \emptyset$ and $I \cap Body^-(r) = \emptyset$. We prove that $(I,J) \models_\beta r$, considering three cases:

Case 1: Assume that $Body^+(r) \not\subseteq I$. Then $(I,J) \not\models_\beta a$, for some atom $a \in Body^+(r)$, and thus $(I,J) \not\models_\beta Body(r)$ which implies $(I,J) \models_\beta r$.

Case 2: Assume that $J \cap Body^-(r) \neq \emptyset$, Then $(I,J) \not\models_\beta \neg a$, for some atom $a \in Body^-(r)$, and thus $(I,J) \not\models_\beta Body(r)$ which implies $(I,J) \models_\beta r$.

Case 3: Assume that $Body^+(r) \subseteq I$ and $J \cap Body^-(r) = \emptyset$. Then, since (I,J) satisfies (a), it also holds that $I \cap Head(r) \neq \emptyset$ and $I \cap Body^-(r) = \emptyset$. From $Body^+(r) \subseteq I$ and $I \cap Body^-(r) = \emptyset$, we conclude that $I \models Body(r)$. Moreover, $I \cap Head(r) \neq \emptyset$ implies $(I,J) \models_\beta Head(r)$. Thus, $(I,J) \models_\beta r$.

By our assumption, one of these three cases applies for (I,J), proving the claim.

($\Rightarrow$) Suppose that $(I,J) \models_\beta r$. We prove that (I,J) satisfies (a), distinguishing two cases:

Case 1: Assume that $(I,J) \not\models_\beta Body(r)$. Then either $(I,J) \not\models_\beta a$, for some atom $a \in Body^+(r)$, or $(I,J) \not\models_\beta \neg a$, for some atom $a \in Body^-(r)$. Hence, $Body^+(r) \not\subseteq I$ or $J \cap Body^-(r) \neq \emptyset$, which implies that (I,J) satisfies (a).

Case 2: Assume that $(I,J) \models_\beta Head(r)$ and $I \models Body(r)$. Then $I \cap Head(r) \neq \emptyset$ and $I \cap Body^-(r) = \emptyset$, and thus (I,J) satisfies (a).

By our assumption, one of the two cases applies for (I,J), which proves the claim. □

To every bi-model of a program P, we associate a corresponding interpretation $(I,J)^K$ over Σ^K by $(I,J)^K = I \cup \{Ka \mid a \in J\}$. Conversely, given an interpretation I^K over Σ^K its associated bi-interpretation $\beta(I^K)$ is given by $(I^K \cap \Sigma, \{a \mid Ka \in I^K\})$.

In order to relate these constructions to models of the epistemic transformation, which builds on additional atoms of the form $\lambda_{r,i}$, we construct an interpretation $(I,J)^{K,P}$ of P^K from a given bi-interpretation (I,J) of P as:

$$(I,J)^{K,P} = (I,J)^K \cup \left\{ \begin{array}{l} \lambda_{r,i} \mid r \in P, Body^-(r) \neq \emptyset, a_i \in I, \\ \qquad I \models Body(r), J \models Body(r) \backslash Body^+(r) \end{array} \right\},$$

where r is of the form (2.2).

Proposition 5 *Let P be a program over Σ. Then,*

(1) if (I,J) is a bi-model of P, then $(I,J)^{K,P} \models P^K$;

(2) if $M \models P^K$ then $\beta(M \cap \Sigma^K)$ is a bi-model of P.

$\circ$

Proof of Proposition 5. Let P be a program over Σ.

Part (1). First, let (I,J) be a bi-model of P. We prove that $(I,J)^{K,P} \models P^K$.

Towards a contradiction assume the contrary. Then there exists a rule r' in P^K, such that $(I,J)^{K,P} \not\models r'$. Suppose that r' is not transformed, i.e., $r' \in P$ and $Body^-(r') = \emptyset$. Since $(I,J) \models_\beta r'$, by Proposition 4 we conclude that $Body^+(r') \subseteq I$ implies $I \cap Head(r') \neq \emptyset$ (recall that $Body^-(r') = \emptyset$). By construction $(I,J)^{K,P}$ restricted to Σ coincides with I. Therefore, $Body^+(r') \subseteq (I,J)^{K,P}$ implies $(I,J)^{K,P} \cap Head(r') \neq \emptyset$, i.e., $(I,J)^{K,P} \models r'$, a contradiction.

Next, suppose that r' is obtained by the epistemic transformation of a corresponding rule $r \in P$ of the form (2.2), and consider the following cases:

– r' is of the form (7.1): then $\{b_1,\ldots,b_m\} \subseteq (I,J)^{K,P}$, which implies $Body^+(r) \subseteq I$. Moreover, $Head(r') \cap (I,J)^{K,P} = \emptyset$ by the assumption that $(I,J)^{K,P} \not\models r'$. By construction of $(I,J)^{K,P}$, this implies $J \cap Body^-(r) = \emptyset$. Since $(I,J) \models_\beta r$, we also conclude that $I \cap Head(r) \neq \emptyset$ and that $I \cap Body^-(r) = \emptyset$. Consequently, $J \models Body(r) \backslash Body^+(r)$, $a_i \in I$ for some $a_i \in Head(r)$, and $I \models Body(r)$. Note also, that $Body^-(r) \neq \emptyset$ by definition of the epistemic transformation. According to the construction of $(I,J)^{K,P}$, it follows that $\lambda_{r,i} \in (I,J)^{K,P}$, a contradiction to $Head(r') \cap (I,J)^{K,P} = \emptyset$.

– r' is of the form (7.2): in this case, $(I,J)^{K,P} \not\models r'$ implies $\lambda_{r,i} \in (I,J)^{K,P}$ and $a_i \notin (I,J)^{K,P}$. However, by construction $\lambda_{r,i} \in (I,J)^{K,P}$ implies $a_i \in I$; from the latter, again by construction, we conclude $a_i \in (I,J)^{K,P}$, a contradiction.

– r' is of the form (7.3): in this case, $(I,J)^{K,P} \not\models r'$ implies $\lambda_{r,i} \in (I,J)^{K,P}$ and $b_j \in (I,J)^{K,P}$. Note that $b_j \in (I,J)^{K,P}$ iff $b_j \in I$. A consequence of the latter is that $I \not\models Body(r)$, contradicting a requirement for $\lambda_{r,i} \in (I,J)^{K,P}$ (cf. the construction of $(I,J)^{K,P}$).

– r' is of the form (7.4): by the assumption that $(I,J)^{K,P} \not\models r'$, it holds that $\lambda_{r,k} \in (I,J)^{K,P}$ and $a_i \in (I,J)^{K,P}$, but $\lambda_{r,i} \notin (I,J)^{K,P}$. From the latter we conclude, by the construction of $(I,J)^{K,P}$, that $a_i \notin I$, since all other requirements for the inclusion of $\lambda_{r,i}$ (i.e., $r \in P$, $Body^-(r) \neq \emptyset$, $I \models Body(r)$, and $J \models Body^-(r)$) must be satisfied as witnessed by $\lambda_{r,k} \in (I,J)^{K,P}$. However, if $a_i \notin I$, then $a_i \notin (I,J)^{K,P}$ (again by construction), contradiction.

This concludes the proof of the fact that if (I,J) is a bi-model of P, then $(I,J)^{K,P} \models P^K$.

Part (2). Let M be a model of P^K. We prove that $\beta(M \cap \Sigma^K) = (I,J)$ is a bi-model of P. Note that by construction $I = M \cap \Sigma$ and $J = \{a \mid Ka \in M\}$. First, we consider any rule r in P such that $Body^-(r) = \emptyset$. Then $r \in P^K$, $J \cap Body^-(r) = \emptyset$ and $I \cap Body^-(r) = \emptyset$. Hence, by Proposition 4, we need to show that $Body^+(r) \subseteq (M \cap \Sigma)$ implies $(M \cap \Sigma) \cap Head(r) \neq \emptyset$. Since $r \in P^K$, this follows from the assumption, i.e., $M \models P^K$ implies $M \models r$, and therefore if $Body^+(r) \subseteq M$, then $M \cap Head(r) \neq \emptyset$. Since r is over Σ, this proves the claim for all $r \in P$ such that $Body^-(r) = \emptyset$.

It remains to show that $(I,J) \models_\beta r$ for all $r \in P$ such that $Body^-(r) \neq \emptyset$. Towards a contradiction assume that this is not the case, i.e., (i) $Body^+(r) \subseteq (M \cap \Sigma)$, (ii) $J \cap Body^-(r) = \emptyset$, and either (iii) $(M \cap \Sigma) \cap Head(r) = \emptyset$ or (iv) $(M \cap \Sigma) \cap Body^-(r) \neq \emptyset$ hold for some $r \in P$ of the form (2.2), such that $Body^-(r) \neq \emptyset$. Conditions (i) and (ii), together with $M \models P^K$, imply that $\lambda_{r,i}$ is in M, for some $1 \leq i \leq l$ (cf. the rule of the form (7.1) in the epistemic transformation of r). Consequently, a_i is in M (cf. the corresponding rule of the form (7.2) in the epistemic transformation of r), and hence $a_i \in (M \cap \Sigma)$. This rules out (iii), so (iv) must hold, i.e., $b_j \in (M \cap \Sigma)$, for some $m+1 \leq j \leq n$. But then, M satisfies the body of a constraint in P^K (cf. the corresponding rule of the form (7.3) in the epistemic transformation of r), contradicting $M \models P^K$. This proves that there exists no $r \in P$ such that $Body^-(r) \neq \emptyset$ and $(I,J) \not\models_\beta r$, and thus concludes our proof of $(I,J) \models_\beta r$. Since $r \in P$ was arbitrary, it follows that $\beta(M \cap \Sigma^K)$ is a bi-model of P. □

Based on bi-models, a 1-1 characterisation of semi-stable models succeeds imposing suitable minimality criteria.

Theorem 7.3.1 *Let P be a program over Σ. Then,*

(1) if (I,J) is a bi-model of P such that

(i) *$(I',J) \not\models_\beta P$, for all $I' \subset I$,*

(ii) *$(I,J') \not\models_\beta P$, for all $J' \subset J$, and*

(iii) *there is no bi-model (I',J') of P that satisfies (i) and $J' \setminus I' \subset J \setminus I$,*

then $(I,J)^K \in \mathcal{SST}(P)$;

(2) if $I^K \in \mathcal{SST}(P)$, then $\beta(I^K)$ is a bi-model of P that satisfies (i)-(iii).

○

Proof of Theorem 7.3.1. Let P be a program over Σ. The proof uses the following lemmas.

Lemma 8 *If $M \in \mathcal{AS}(P^K)$, then $\beta(M \cap \Sigma^K)$ satisfies (i).* ○

Proof of Lemma 8. Towards a contradiction assume that $M \in \mathcal{AS}(P^K)$ and $\beta(M \cap \Sigma^K) = (I,J)$ does not satisfy (i). Then, there exists a bi-model (I',J) of P, such that $I' \subset I$. By Proposition 5, $(I',J)^{K,P} \models P^K$. Note that $(I',J)^K \subset (M \cap \Sigma^K)$. Let $S' = \{\lambda_{r,i} \mid \lambda_{r,i} \in (I',J)^{K,P}\}$ and let $S = \{\lambda_{r,i} \mid \lambda_{r,i} \in M\}$. We show that $S' \subseteq S$. Suppose that this is not the case and assume that $\lambda_{r,i} \in S'$ and $\lambda_{r,i} \notin S$, for some $r \in P$ of the form (2.2) and $1 \leq i \leq l$. By the construction of $(I',J)^{K,P}$, we conclude that $a_i \in I'$, $I' \models Body(r)$, and $J \models Body^-(r)$. Since $I' \subset I$, it also holds that $a_i \in I$ and that $I \models Body^+(r)$. Consider the rule of the form (7.1) of the epistemic transformation of r. We conclude that $\{b_1,\ldots,b_m\} \subseteq M$ (due to $I \models Body^+(r)$), and that $M \not\models Kb_{m+1} \vee \ldots \vee Kb_n$ (due to $J \models Body^-(r)$). But $M \models P^K$, hence $\lambda_{r,k}$ is in M, for some $1 \leq k \leq l$. However, considering the corresponding rule of the form (7.4) of the epistemic transformation of r, we also conclude that $\lambda_{r,i} \in M$, a contradiction. Therefore $S' \subseteq S$ holds, and since $(I',J)^K \subset (M \cap \Sigma^K)$, we conclude that $(I',J)^{K,P} \subset M$. The latter contradicts the assumption that M is an answer set, i.e., a minimal model, of P^K. This concludes the proof of the lemma. □

Next, we prove:

Lemma 9 *If (I,J) is a bi-model of P that satisfies (i) and (ii), then there exists some $M \in \mathcal{AS}(P^K)$, such that $\beta(M \cap \Sigma^K) = (I,J)$.* ∘

Proof of Lemma 9. Let (I,J) be a bi-model of P that satisfies (i) and (ii). If $(I,J)^{K,P} \in \mathcal{AS}(P^K)$, then (c) holds since $\beta((I,J)^{K,P} \cap \Sigma^K) = (I,J)$. If $(I,J)^{K,P} \notin \mathcal{AS}(P^K)$, then there exists a minimal model, i.e., an answer set, M' of P^K, such that $M' \subset (I,J)^{K,P}$. Let $(I',J') = \beta(M' \cap \Sigma^K)$. Then $I' \subseteq I$ and $J' \subseteq J$ holds by construction and the fact that $M' \subset (I,J)^{K,P}$. Towards a contradiction, assume that $I' \subset I$. We show that then (I',J) is a bi-model of P. Suppose that (I',J) is not a bi-model of P. Then, by Proposition 4, there exists $r \in P$, such that $Body^+(r) \subseteq I'$, $J \cap Body^-(r) = \emptyset$, and either $I' \cap Head(r) = \emptyset$ or $I' \cap Body^-(r) \neq \emptyset$. Note that $Body^+(r) \subseteq I'$ implies $Body^+(r) \subseteq I$, and since (I,J) is a bi-model of P, we conclude $I \cap Head(r) \neq \emptyset$ and $I \cap Body^-(r) = \emptyset$. The latter implies $I' \cap Body^-(r) = \emptyset$, hence $I' \cap Head(r) = \emptyset$ holds. If $Body^-(r) = \emptyset$, then r is in P^K and $M' \not\models r$, contradiction. Thus, $Body^-(r) \neq \emptyset$. However, in this case the epistemic transformation of r is in P^K. Since $J \cap Body^-(r) = \emptyset$ and $J' \subseteq J$ together imply $J' \cap Body^-(r) = \emptyset$, we conclude that for the rule of the form (7.1) of the epistemic transformation of r, it holds that $\{b_1,\ldots,b_m\} \subseteq M'$ (due to $Body^+(r) \subseteq I'$), and that $M' \not\models Kb_{m+1} \vee \ldots \vee Kb_n$ (due to $J' \cap Body^-(r) = \emptyset$). Moreover $M' \models P^K$, hence $\lambda_{r,i}$ is in M', for some $1 \leq i \leq l$. Considering the corresponding rule of the form (7.2) of the epistemic transformation of r, we also conclude that $a_i \in M'$, a contradiction to $I' \cap Head(r) = \emptyset$. This proves that (I',J) is a bi-model of P, and thus contradicts the assumption that (I,J) satisfies (i). Consequently, $I' = I$. Now if $J' \subset J$, then we obtain a contradiction with the assumption that (I,J) satisfies (ii). Therefore also $J' = J$, which concludes the proof of the Lemma. □

The proof of the theorem is then as follows.

Part (1). Let (I,J) be a bi-model of P that satisfies (i)-(iii). We prove that $(I,J)^K \in \mathcal{SST}(P)$. By Lemma 9, we conclude that there exists some $M \in \mathcal{AS}(P^K)$ such that $\beta(M \cap \Sigma^K) = (I,J)$. It remains to show that M is maximal canonical. Towards a contradiction assume the contrary. Then, there exists $M' \in \mathcal{AS}(P^K)$ such that $gapM' \subset gapM$. Let $(I',J') = \beta(M' \cap \Sigma^K)$. By Lemma 8, (I',J') satisfies (i), and by construction since $gapM' \subset gapM$, it holds that $J' \setminus I' \subset J \setminus I$. However, this contradicts the assumption that (I,J) satisfies (iii). Therefore, M is maximal canonical, and hence $(I,J)^K \in \mathcal{SST}(P)$.

Part (2). Let $I^K \in \mathcal{SST}(P)$. We show that $\beta(I^K)$ is a bi-model of P that satisfies (i)-(iii). Let $(I,J) = \beta(I^K)$ and let M be a maximal canonical answer set of P^K corresponding to I^K. Then, $\beta(M \cap \Sigma^K) = (I,J)$ by construction, and (I,J) satisfies (i) by Lemma 8.

Towards a contradiction first assume that (I,J) does not satisfy (iii). Then there exists a bi-model (I',J') of P such that (I',J') satisfies (i) and $J' \setminus I' \subset J \setminus I$. Let $M' = (I',J')^{K,P}$ and note that if $M' \in \mathcal{AS}(P^K)$, we arrive at a contradiction to $M \in mc(\mathcal{AS}(P^K))$, since $gapM' \subset gapM$. Thus, there exists $M'' \in \mathcal{AS}(P^K)$, such that $M'' \subset M'$. Let $(I'',J'') = \beta(M'' \cap \Sigma^K)$. We show that (I'',J') is a bi-model of P, and thus by (i) it follows that $I'' = I'$.

Towards a contradiction, suppose that (I'',J') is not a bi-model of P. Then, by Proposition 4, there exists $r \in P$, such that $Body^+(r) \subseteq I''$, $J' \cap Body^-(r) = \emptyset$, and either $I'' \cap Head(r) = \emptyset$ or $I'' \cap Body^-(r) \neq \emptyset$. Note that $Body^+(r) \subseteq I''$ implies $Body^+(r) \subseteq I'$, and since (I',J') is a bi-model of P, we conclude $I' \cap Head(r) \neq \emptyset$ and $I' \cap Body^-(r) = \emptyset$. The latter implies $I'' \cap Body^-(r) = \emptyset$, hence $I'' \cap Head(r) = \emptyset$ holds. If $Body^-(r) = \emptyset$, then r is in P^K and $M'' \not\models r$, contradiction. Thus, $Body^-(r) \neq \emptyset$. However, in this case the epistemic transformation of r is in P^K. Since $J' \cap Body^-(r) = \emptyset$ and $J'' \subseteq J'$ together imply $J'' \cap Body^-(r) = \emptyset$, we conclude that for the rule of the form (3) of the epistemic transformation of r, it holds that $\{b_1,\ldots,b_m\} \subseteq M''$ (due to $Body^+(r) \subseteq I''$), and that $M'' \not\models Kb_{m+1} \vee \ldots \vee Kb_n$ (due to $J'' \cap Body^-(r) = \emptyset$). Moreover $M'' \models P^K$, hence $\lambda_{r,i}$ is in M'', for some $1 \leq i \leq l$. Considering the corresponding rule of the form (4) of the epistemic transformation of r, we also conclude that $a_i \in M''$, a contradiction to $I'' \cap Head(r) = \emptyset$.

This proves that (I'',J') is a bi-model of P. From the assumption that (I',J') satisfies (i), it follows that $I'' = I'$. Therefore $gapM'' \subseteq gapM'$ holds, which implies $gapM'' \subset gapM$, a contradiction to $M \in mc(\mathcal{AS}(P^K))$. This proves (I,J) satisfies (iii).

Next assume that (I,J) does not satisfy (ii). Then, there exists a bi-model (I,J') of P, such that $J' \subset J$. We show that (I,J') satisfies (i). Otherwise, there exists a bi-model (I',J') of P, such that $I' \subset I$; but then also (I',J) is a bi-model of P. To see the latter, assume that there exists a rule $r \in P$, such that $Body(r) \subseteq I'$, $J \cap Body^-(r) = \emptyset$ and either $I' \cap Head(r) = \emptyset$ or $I' \cap Body^-(r) \neq \emptyset$. Since $J' \subset J$, it then also holds that $J' \cap Body^-(r) = \emptyset$. This contradicts the assumption that (I',J') is a bi-model of P, hence $(I',J) \models_\beta P$. The latter is a contradiction to the assumption that (I,J) satisfies (i), proving that (I,J') satisfies (i). Since (I,J) satisfies (iii), we conclude

that $J' \setminus I = J \setminus I$. Now let $S' = \{\lambda_{r,i} \mid \lambda_{r,i} \in (I,J')^{K,P}\}$ and let $S = \{\lambda_{r,i} \mid \lambda_{r,i} \in M\}$. It holds that $S' \not\subseteq S$ (otherwise $(I,J')^{K,P} \subset M$, a contradiction to $M \in \mathcal{AS}(P^K)$), i.e., there exists $r \in P$ of the form (1) and $1 \leq i \leq l$, such that $\lambda_{r,i} \in S$ and $\lambda_{r,i} \notin S'$. From the former, since M is a minimal model of P^K, we conclude that $I \models Body^+(r)$, $a_i \in I$, and $J \cap Body^-(r) = \emptyset$. Since $J' \subset J$, also $J' \cap Body^-(r) = \emptyset$. This implies that $\lambda_{r,k} \in S'$, for some $1 \leq k \neq i \leq l$ (otherwise $(I,J')^{K,P}$ does not satisfy the rule of form (3) corresponding to r in P^K, a contradiction to $(I,J')^{K,P} \models P^K$). However, since $a_i \in I$, and thus $a_i \in (I,J')^{K,P}$, and since $\lambda_{r,k} \in (I,J')^{K,P}$, we conclude that $\lambda_{r,i} \in (I,J')^{K,P}$ (cf. the respective rule of form (6) of the epistemic transformation of r). This contradicts $\lambda_{r,i} \notin S'$, and thus proves that (I,J) satisfies (ii). □

Intuitively, Conditions (i) and (ii) filter bi-models that uniquely correspond to (some but not all) answer sets of P^K: due to minimality every answer set satisfies (i); there may be answer sets of P^K that do not satisfy (ii), but they are certainly not maximal canonical. Eventually, Condition (iii) ensures that maximal cononical answer sets are selected. More formally, the proof of this theorem builds on the following relationship between bi-models of P and answer sets of P^K.

Corollary 1 *Let P be a program over Σ. If $M \in \mathcal{AS}(P^K)$, then $\beta(M \cap \Sigma^K)$ satisfies (i). If (I,J) is a bi-model of P that satisfies (i) and (ii), then there exists $M \in \mathcal{AS}(P^K)$, such that $\beta(M \cap \Sigma^K) = (I,J)$.*

For illustration consider the following example.

Example 7.3.3 *Let $P = \{a \leftarrow b;\ b \leftarrow not\ b\}$. Its bi-models are all pairs (I,J), where $I \in \{\emptyset, \{a\}, \{a,b\}\}$ and $J \in \{\{b\}, \{a,b\}\}$. Condition (i) of Theorem 7.3.1 holds for bi-models such that $I = \emptyset$, and Condition (ii) holds only-if $J = \{b\}$. Thus, $\{Kb\}$ is the unique semi-stable model of P.* △

The examples given so far also exhibit some anomalies of the semi-stable semantics with respect to basic rationality properties considered in epistemic logics. In particular, *knowledge generalisation* (or *necessitation*, respectively modal axiom **N**) is a basic principle in respective modal logics. For a semi-stable model I^K, it would require that

Property N: $a \in I^K$ implies $Ka \in I^K$, for all $a \in \Sigma$.

This property does not hold as witnessed by Example 7.3.2.

Another basic requirement is the *distribution axiom* (modal axiom **K**). Assuming that we believe the rules of a given program (which might also be seen as the consequence of adopting knowledge generalisation) the distribution property can be paraphrased for a rule of the form (2.2) as follows:

Property K: If $I^K \models Kb_1 \wedge \ldots \wedge Kb_m$ and
$I^K \not\models Kb_{m+1} \vee \ldots \vee Kb_n$, then $I^K \models Ka_1 \vee \ldots \vee Ka_l$.

Note that this does not hold for rule $a \leftarrow b$ in Example 7.3.3.

7.4 An Alternative Paracoherent Semantics

In this section we define and characterise an alternative paracoherent semantics which we call semi-equilibrium semantics (for reasons which will become clear immediately). The aim for semi-equilibrium models is to enforce Properties **N** and **K** on them. Let us start considering bi-models of a program P, that satisfy these properties. It turns out that such structures are exactly given by HT-models.

Proposition 6 *Let P be a program over Σ. Then,*

(1) if (I,J) is a bi-model of P, such that $(I,J)^K$ satisfies Property **N** *and Property* **K**, *for all $r \in P$, then (I,J) is an HT-model of P;*

(2) if (H,T) is an HT-model of P, then $(H,T)^K$ satisfies Property **N** *and Property* **K**, *for all $r \in P$.*

∘

Proof of Proposition 6. Let P be a program over Σ.

Part (1). Let (I,J) be a bi-model of P, such that $(I,J)^K$ satisfies Property **N** and Property **K**, for all $r \in P$. We show that (I,J) is an HT-model of P. Since $(I,J)^K$ satisfies Property **N**, it holds that $a \in I$ implies $a \in J$, therefore $I \subseteq J$, i.e., (I,J) is an HT-interpretation. For every rule $r \in P$, $(I,J) \models_\beta r$ implies $(I,J) \not\models_\beta Body(r)$, or $(I,J) \models_\beta Head(r)$ and $I \models Body(r)$.

First suppose that $(I,J) \not\models_\beta Body(r)$. Then $(I,J) \not\models Body(r)$ (note that for a conjunction of literals, such as $Body(r)$, the satisfaction relations do not differ). Moreover, since $(I,J)^K$ satisfies Property **K** for r, it holds that $J \models r$. To see the latter, let Kr denote the rule obtained from r by replacing every $a \in \Sigma$ occurring in r by Ka, and let KJ denote the set $\{Ka \in (I,J)^K \mid a \in \Sigma\}$. Then, $(I,J)^K$ satisfies Property **K** for r iff $KJ \models Kr$. Observing that $KJ = \{Ka \mid a \in J\}$, we conclude that $J \models r$. This proves $(I,J) \models r$, if $(I,J) \not\models_\beta Body(r)$.

Next assume that $(I,J) \models_\beta Head(r)$ and $I \models Body(r)$. We conclude that $(I,J) \models Head(r)$ (the satisfaction relations also coincide for disjunctions of atoms). As $(I,J)^K$ satisfies Property **K** for r, it follows $J \models r$. This proves $(I,J) \models r$, for every $r \in P$; in other words, (I,J) is an HT-model of P.

Part (2). Let (H,T) be an HT-model of P. We show that $(H,T)^K$ satisfies Property **N** and Property **K**, for all $r \in P$. As a consequence of $H \subseteq T$, for every $a \in (H,T)^K$ such that $a \in \Sigma$, it also holds that $Ka \in (H,T)^K$, i.e., $(H,T)^K$ satisfies Property **N**. Moreover, $(H,T) \models P$ implies $T \models r$, for all $r \in P$. Let $KT = \{Ka \mid a \in T\}$ and let Kr as above; $T \models r$ implies $KT \models Kr$, for all $r \in P$. By construction of $(H,T)^K$ and definition of Property **K** for r, we conclude that $(H,T)^K$ satisfies Property **K** for all $r \in P$. □

In order to define semi-equilibrium models, we follow the basic idea of the semi-stable semantics and select subset minimal models that are maximal canonical. Let us

define $HT^K(P) = \{(H,T)^K \mid (H,T) \models P\}$ and denote by $\mathcal{MM}(HT^K(P))$ its minimal elements with respect to subset inclusion.

Definition 7.4.1 (Semi-Equilibrium Models) *Let P be a program over Σ. An interpretation I^K over Σ^K is a semi-equilibrium model of P iff $I^K \in mc(\mathcal{MM}(HT^K(P)))$. The set of semi-equilibrium models of P is denoted by $\mathcal{SEQ}(P)$.* ▲

A model-theoretic characterisation for this semantics is obtained as before, replacing bi-models by HT-models and dropping Condition (ii). Intuitively, Condition (ii) is not needed as it is subsumed by Condition (iii) (i.e., Condition (ii′) below) if Property **N** and Condition (i) hold.

To formulate the result, we extend the notion of *gap* from Σ^K-interpretations to HT-interpretations as follows. For any HT-interpretation (X,Y), let $gap(X,Y) = Y \setminus X$, i.e., $gap(X,Y) = gap(\beta((X,Y)^K)) = \{a \mid Ka \in gap((X,Y)^K)\}$.

Theorem 7.4.1 *Let P be a program over Σ. Then,*

(1) If (H,T) is an HT-model of P such that

(i′) *$(H',T) \not\models P$, for all $H' \subset H$, and*

(ii′) *there is no HT-model (H',T') of P that satisfies* (i′) *and $T' \setminus H' \subset T \setminus H$,*

then $(H,T)^K \in \mathcal{SEQ}(P)$;

(2) if $I^K \in \mathcal{SEQ}(P)$, then $\beta(I^K)$ is an HT-model of P that satisfies (i′) *and* (ii′).

○

Proof of Theorem 7.4.1. Let P be a program over Σ.

Part (1). Let (H,T) be an HT-model of P that satisfies (i′) and (ii′). We show that $(H,T)^K \in \mathcal{SEQ}(P)$. Towards a contradiction, first assume that $(H,T)^K \notin \mathcal{MM}(HT^K(P))$. Then, there exists an HT-model (H',T') of P, such that $H' \subseteq H$, $T' \subseteq T$, and at least one of the inclusions is strict. Suppose that $H' \subset H$. Then (H',T) is an HT-model of P (by a well-known property of HT), a contradiction to the assumption that (H,T) satisfies (i′). Hence, $H' = H$ and $T' \subset T$ must hold. Moreover, by the same argument (H',T') also satisfies (i′). But, since $T' \setminus H' \subset T \setminus H$, this is in contradiction to the assumption that (H,T) satisfies (ii′). Consequently, $(H,T)^K \in \mathcal{MM}(HT^K(P))$.

We continue the indirect proof assuming that $(H,T)^K \notin mc(\mathcal{MM}(HT^K(P)))$, i.e., there exists an HT-model (H',T') of P, such that $T' \setminus H' \subset T \setminus H$ and $(H',T')^K \in \mathcal{MM}(HT^K(P))$. The latter obviously implies that (H',T') satisfies (i′).

Again, this contradicts that (H,T) satisfies (ii′), which proves that $(H,T)^K \in \mathcal{SEQ}(P)$.

Part (2). Let $I^K \in \mathcal{SEQ}(P)$. We show that $\beta(I^K)$ is an HT-model of P that satisfies (i′) and (ii′). Let $\beta(I^K) = (H,T)$. Towards a contradiction first assume that

(H,T) is not an HT-model of P. Then by the definition of $\mathcal{SEQ}(P)$, and the fact that I^K uniquely corresponds to sets H and T, we conclude that $I^K \notin mc(\mathcal{MM}(HT^K(P)))$, i.e., $I^K \notin \mathcal{SEQ}(P)$; contradiction.

Next, suppose that (H,T) does not satisfy (i′). Then, $I^K \notin \mathcal{MM}(HT^K(P))$, as witnessed by $(H',T)^K$ for an HT-model (H',T) such that $H' \subset H$, which exists if (H,T) does not satisfy (i′). Therefore, $I^K \notin mc(\mathcal{MM}(HT^K(P)))$, i.e., $I^K \notin \mathcal{SEQ}(P)$; contradiction.

Eventually assume that (H,T) does not satisfy (ii′). Then, $I^K \notin mc(\mathcal{MM}(HT^K(P)))$, as witnessed by $(H',T')^K$ for an HT-model (H',T'), such that $T' \setminus H' \subset T \setminus H$ and (H',T') satisfies (i′)—note that (H',T') exists if (H,T) does not satisfy (ii′). To see that $(H',T')^K$ is a witness for $I^K \notin mc(\mathcal{MM}(HT^K(P)))$, observe that either $(H',T')^K \in \mathcal{MM}(HT^K(P))$ or there exists an HT-model (H',T''), such that $(H',T'')^K \in \mathcal{MM}(HT^K(P))$ and $T'' \subset T'$ (which implies $T'' \setminus H' \subset T' \setminus H' \subset T \setminus H$).

This proves that $I^K \notin \mathcal{SEQ}(P)$, again a contradiction. This concludes the proof that $\beta(I^K)$ is an HT-model of P that satisfies (i′) and (ii′). □

Alternatively, semi-equilibrium models may be computed as maximal canonical answer sets, i.e., equilibrium models, of an extension of the epistemic program transformation.

Definition 7.4.2 (P^{HT}) *Let P be a program over Σ. Then its epistemic HT-transformation P^{HT} is defined as the union of P^K with the set of rules:*

$$Ka \leftarrow a,$$
$$Ka_1 \vee \ldots \vee Ka_l \vee Kb_{m+1} \vee \ldots \vee Kb_n \leftarrow Kb_1, \ldots, Kb_m,$$

for $a \in \Sigma$, respectively for every rule $r \in P$ of the form (2.2). ▲

The extensions of the transformation naturally ensure Properties **N** and **K** on its models and its maximal canonical answer sets coincide with semi-equilibrium models.

Theorem 7.4.2 *Let P be a program over Σ, and let I^K be an interpretation over Σ^K. Then, $I^K \in \mathcal{SEQ}(P)$ if and only-if $I^K \in \{M \cap \Sigma^K \mid M \in mc(\mathcal{AS}(P^{HT}))\}$.* ○

Proof of Theorem 7.4.2. Let P be a program over Σ, and let I^K be an interpretation over Σ^K. The proof of this theorem uses the following auxiliary lemmas which we state, and prove, before proceeding.

Lemma 10 *If $M \models P^{HT}$, then $\beta(M \cap \Sigma^K)$ is an HT-model of P.* ○

Proof of Lemma 10. Let $(I,J) = \beta(M \cap \Sigma^K)$. Since $M \models P^K$, (I,J) is a bi-model of P by Proposition 5. Moreover, $M \cap \Sigma^K = (I,J)^K$ and $(I,J)^K$ satisfies Property **N**, otherwise there is an atom $a \in M$ such that $Ka \notin M$, a contradiction to $M \models Ka \leftarrow a$. Also, $(I,J)^K$ satisfies Property **K** for all $r \in P$; otherwise, if Property **K** does not hold for some $r \in P$ of the form (2.2), then $M \models Kb_1 \wedge \ldots \wedge Kb_m$ and $M \not\models Ka_1 \vee \ldots \vee Ka_l \vee$

$Kb_{m+1} \vee \ldots \vee Kb_n$, i.e., $M \not\models P^{HT}$; contradiction. Therefore, by Proposition 6, (I,J) is a HT-model of P. □

Next, we prove two Lemmas before proceeding with the proof of the main theorem:

Lemma 11 *If (H,T) is an HT-model of P, then $(H,T)^{K,P} \models P^{HT}$.* ○

Proof of Lemma 11. Note that every HT-model of P is a bi-model of P. Assume the contrary; then $(H,T) \models r$ and $(H,T) \not\models_\beta r$, for some $r \in P$. Then, $H \not\models Body(r)$, while $(H,T) \models Body(r)$, must hold. However, $(H,T) \models Body(r)$ implies $Body^+(r) \subseteq H$ and $Body^-(r) \cap H = \emptyset$, and therefore $H \models Body(r)$; contradiction. This proves that (H,T) is a bi-model of P. Consequently, $(H,T)^{K,P} \models P^K$ by Proposition 5.

Moreover, since (H,T) is an HT-model, $(H,T)^K$ satisfies Property **N** (and Property **K** for all $r \in P$) by Proposition 6. Because $(H,T)^{K,P} \cap \Sigma^K = (H,T)^K$, this implies that $(H,T)^{K,P} \models r$, for all rules of the form $Ka \leftarrow a$ in $P^{HT} \setminus P^K$ (this is an obvious consequence of Property **N**).

For the remaining rules r in $P^{HT} \setminus P^K$, $(H,T)^{K,P} \models r$ is a simple consequence of $T \models P$. This proves $(H,T)^{K,P} \models P^{HT}$. □

Now let $M \in \mathcal{AS}(P^{HT})$. We show that

Lemma 12 *For every $M \in \mathcal{AS}(P^{HT})$, $\beta(M \cap \Sigma^K)$ satisfies* (i′) *in Theorem 7.4.1.* ○

Proof of Lemma 12. Towards a contradiction assume the contrary. Then there exists an HT-model (H',T) of P such that $H' \subset H$. Note that $M \in \mathcal{AS}(P^{HT})$ implies $M = \beta(M \cap \Sigma^K)^{K,P}$. Since the latter is a model of P^{HT} by Lemma 11, M must be a subset thereof; however it obviously cannot be a strict subset on Σ^K. By construction of $\beta(M \cap \Sigma^K)^{K,P}$ and the rules of form (6) of the epistemic transformation, we also conclude that $\lambda_{r,i} \in \beta(M \cap \Sigma^K)^{K,P}$ implies $\lambda_{r,i} \in M$, for any $r \in P$ of the form (2.2) and $1 \leq i \leq l$. This proves $M = \beta(M \cap \Sigma^K)^{K,P}$.

Now consider $M' = (H',T)^{K,P}$. Then, $M' \subset M$ by construction, and $M' \models P^{HT}$ by Lemma 11. This is a contradiction to the assumption that $M \in \mathcal{AS}(P^{HT})$, and thus proves that $\beta(M \cap \Sigma^K)$ satisfies (i′). □

Lemma 13 *For every HT-model (H,T) of P that satisfies* (i′) *of Theorem 7.4.1, there exists some $M \in \mathcal{AS}(P^{HT})$ such that $gapM \subseteq gap(H,T)^K$.* ○

Proof of Lemma 13. Since $(H,T)^{K,P} \models P^{HT}$ by Lemma 11, there exists $M \in \mathcal{AS}(P^{HT})$, such that $M \subseteq (H,T)^{K,P}$. To prove the lemma, it suffices to show that $M \cap \Sigma = H$. Assume the contrary; then by (d) there exists an HT-model (H',T') of P, such that $H' \subset H$ and $T' \subseteq T$. However, then $(H',T) \models P$, which contradicts the assumption that (H,T) satisfies (i′). □

We now proceed with the proof of the theorem, which is then as follows.

($\Leftarrow$) Suppose that $I^K \in \{M \cap \Sigma^K \mid M \in mc(\mathcal{AS}(P^{HT}))\}$. We prove $I^K \in \mathcal{SEQ}(P)$ via Theorem 7.4.1. Let $M \in mc(\mathcal{AS}(P^{HT}))$, such that $I^K = M \cap \Sigma^K$, and let $(I,J) = \beta(M \cap \Sigma^K)$. Then, (I,J) is an HT-model of P by Lemma 10 and (I,J) satisfies (i$'$) in Theorem 7.4.1 by Lemma 12. We prove that (I,J) satisfies (ii$'$) in Theorem 7.4.1.

Towards a contradiction, assume that this is not the case, then there exists an HT-model (H,T) of P, such that $T \setminus H \subset J \setminus I$ and (H,T) satisfies (i$'$). According to Lemma 13, there exists $M' \in \mathcal{AS}(P^{HT})$, such that $gapM' \subseteq gap(H,T)^K$, which implies $gapM' \subset gapM$ due to $T \setminus H \subset J \setminus I$.

This contradicts the assumption that $M \in mc(\mathcal{AS}(P^{HT}))$, and thus proves that (I,J) satisfies (ii$'$) in Theorem 7.4.1. We conclude that $I^K \in \mathcal{SEQ}(P)$.

($\Rightarrow$) Suppose that $I^K \in \mathcal{SEQ}(P)$. We prove $I^K \in \{M \cap \Sigma^K \mid M \in mc(\mathcal{AS}(P^{HT}))\}$. Let $(H,T) = \beta(I^K)$. By Theorem 7.4.1, (H,T) is an HT-model of P that satisfies (i$'$) and (ii$'$).

We show that there exists $M \in mc(\mathcal{AS}(P^{HT}))$ such that $\beta(M \cap \Sigma^K) = (H,T)$. Since $(H,T)^{K,P} \models P^{HT}$, there exists $M \in \mathcal{AS}(P^{HT})$ such that $M \subseteq (H,T)^{K,P}$. Since (H,T) satisfies (i$'$), it holds that $M \cap \Sigma = H$. Moreover, $M \cap \Sigma^K \subset (H,T)^K$ contradicts the fact that (H,T) satisfies (ii$'$), because then $\beta(M \cap \Sigma^K) = (H,T')$ is an HT-model of P, such that $T' \setminus H \subset T \setminus H$ and (H,T') satisfies (i$'$) due to Lemma 12. Hence, $\beta(M \cap \Sigma^K) = (H,T)$.

It remains to show that $M \in mc(\mathcal{AS}(P^{HT}))$. If this is not the case, then some HT-model (H',T') of P exists such that $T' \setminus H' \subset T \setminus H$. Since $(H',T') = \beta(M' \cap \Sigma^K)$ for some $M' \in \mathcal{AS}(P^{HT})$, we conclude by Lemma 12 that (H',T') satisfies (i$'$), which again leads to a contradiction of the fact that (H,T) satisfies (ii$'$). This proves that $M \in mc(\mathcal{AS}(P^{HT}))$. As $M \cap \Sigma^K = I^K$, we conclude that $I^K \in \{M \cap \Sigma^K \mid M \in mc(\mathcal{AS}(P^{HT}))\}$. □

The resulting semantics is classically coherent.

Proposition 7 *Let P be a program over Σ. If P has a model, then it has a semi-equilibrium model.* ○

Proof of Proposition 7. Let P be a program over Σ. If P has a model M, then (M,M) is an HT-model of P. Therefore $HT^K(P) \neq \emptyset$, which implies $\mathcal{MM}(HT^K(P)) \neq \emptyset$, and thus $mc(\mathcal{MM}(HT^K(P))) \neq \emptyset$. We conclude that $\mathcal{SEQ}(P) \neq \emptyset$, i.e., P has a semi-equilibrium model. □

Another simple property is a 1-to-1 correspondence between answer sets and semi-equilibrium models.

Proposition 8 *Let P be a coherent program over Σ. Then,*

(1) if $Y \in \mathcal{AS}(P)$, then $(Y,Y)^K$ is a semi-equilibrium model of P;

(2) if I^K is a semi-equilibrium model of P then $\beta(I^K)$ is an equilibrium model of P, i.e., $\beta(I^K)$ is of the form (Y,Y) and $Y \in \mathcal{AS}(P)$.

$\circ$

Proof of Proposition 8. Let P be a coherent program over Σ, and let $Y \in \mathcal{AS}(P)$. Then (Y,Y) is an HT-model of P that satisfies (i$'$) in Theorem 7.4.1, since it is in equilibrium. Moreover, it trivially satisfies also (ii$'$) because $Y \setminus Y = \emptyset$. Hence, $(Y,Y)^K \in \mathcal{SEQ}(P)$.

As P is coherent, there exists $(T,T) \in HT(P)$ that satisfies (i$'$) in Theorem 7.4.1 and (trivially) (ii$'$). Hence, $gap I^K = \emptyset$ for all $I^K \in \mathcal{SEQ}(P)$. Moreover, $\beta(I^K)$ is of the form (Y,Y), and $Y \in \mathcal{AS}(P)$. $\square$

For an illustration of the 1-to-1 relationship between answer sets and semi-equilibrium models, let us reconsider Example 7.3.2. Note that this example also gave evidence that semi-stable models do not satisfy Property **N**, which is the case for semi-equilibrium models, however.

Example 7.4.1 *Consider the coherent program of Example 7.3.2. Its unique semi-equilibrium model is $\{a,b,c,Ka,Kb,Kc\}$, corresponding to the single answer set $\{a,b,c\}$.* $\triangle$

As a consequence of Property **K**, semi-equilibrium semantics differs from semi-stable semantics not only with respect to believed consequences.

Example 7.4.2 *Consider the following extension of the program in Example 7.3.3: $P = \{a \leftarrow b;\ b \leftarrow not\ b;\ c \leftarrow not\ a\}$, and compare $\mathcal{SST}(P) = \{\{c,Kb\}\}$ with $\mathcal{SEQ}(P) = \{\{Ka,Kb\}\}$ concerning the knowledge with respect to c.* $\triangle$

7.5 Computational Complexity

We now consider the computational complexity of major reasoning tasks for programs under semi-stable semantics:

a) deciding whether a program P has a semi-stable model,

b) recognising semi-stable models (given M and P), and

c) brave and cautious reasoning from the semi-stable models of a program P, where an atom a is a brave (respectively, cautious) consequence of P with value $v \in \{\bot, \mathbf{bt}, \mathbf{t}\}$, denoted $P \models^v_b a$ (respectively, $P \models^v_c a$) if $I^K(a) = v$ for some (respectively, every) semi-stable model of P.

We also consider these problems for semi-equilibrium models. For brevity, we compactly summarise the results in Table 7.1. In the following, we overview the main results but leave the proofs for being read from the original article Eiter *et al.* (2010b).

Lemma 14 *Given a bi-interpretation (I,J) of a program P, deciding $(I,J) \models_\beta P$ is polynomial.* $\circ$

Problem	normal P	disjunctive P
$\mathcal{SST}(P) \neq \emptyset$?	NP	NP
$I^K \in \mathcal{SST}(P)$?	coNP	Π_2^P
$P \models_b^v a$	Σ_2^P	Σ_3^P
$P \models_c^v a$	Π_2^P	Π_3^P

Table 7.1: Complexity of semi-stable models (completeness results). The same results hold for semi-equilibrium models.

7.5.1 Problem a)

Since P has some semi-stable model iff P has some classical model, the complexity of Problem a) is immediate.

Theorem 7.5.1 *Given a disjunctive program P, deciding whether $\mathcal{SST}(P) \neq \emptyset$ is* NP*-complete. The* NP*-hardness holds already for normal programs P.* ○

However, the problem is trivial, if the program P has no constraints, and is polynomial e.g. if P consists of Horn clauses.

7.5.2 Problem b)

Recognising semi-stable models, however, is more complex than recognising classical models.

Theorem 7.5.2 *Given an interpretation I^K over Σ^K and a program P, deciding if $I^K \in \mathcal{SST}(P)$ is*

(i) coNP*-complete for normal P, and*

(ii) Π_2^P*-complete for disjunctive P.*

○

7.5.3 Problem c)

Brave respectively cautious reasoning from the semi-stable models of programs is one level higher up in the Polynomial Hierarchy than respective reasoning from the stable models. Intuitively, this is because maximal canonicality is an orthogonal selection on top of the stable models.

Theorem 7.5.3 *Given a program P, an atom a, and a value $v \in \{\bot, \mathbf{bt}, \mathbf{t}\}$, deciding whether*

(i) $P \models_b^v a$ *is* Σ_2^P*-complete for normal* P *and* Σ_3^P*-complete for disjunctive* P*;*

(ii) $P \models_c^v a$ *is* Π_2^P*-complete for normal* P *and* Π_3^P*-complete for disjunctive* P*.*

○

7.5.4 Semi-Equilibrium Models

For disjunctive programs, Problems a)-c) have for semi-equilibrium models the same complexity as for semi-stable models; we omit stating here the formal results.

The memberships proofs are similar but use Theorem 7.4.1 instead of Theorem 7.3.1; note that deciding $(H,T) \models P$ is polynomial, and thus checking condition (i') in Theorem 7.4.1 is in coNP. Furthermore, similar hardness proofs work (with slight adaptions).

Also for normal programs P, we get analogous complexity results, since testing condition (i') for such P is polynomial: similarly as for condition (i) in Theorem 7.3.1, we can compute some $H' \subset H$ such that $(H',T) \models P$ (if one exists) as the least model of a set of Horn clauses (more precisely of P^T).

In all cases, we also have matching hardness results. This follows easily from well-known results for answer sets semantics of positive disjunctive programs, cf. Eiter & Gottlob (1995a), which contain, without loss of generality, no facts; we can easily emulate such programs under semi-equilibrium semantics, shifting disjunctions to the rule bodies, and creating constraints in this way.

7.6 Discussion

In this section, we first review some general principles for logic programs with negation, and we then consider the relationship of semi-stable and semi-equilibrium semantics to other semantics of logic programs with negation. Finally, we address some possible extensions of our work.

7.6.1 General Principles

In the context of logic programs with negation, several principles have been identified which a semantics desirably should satisfy. Among them are:

- the *principle of minimal undefinedness* You & Yuan (1994), which says that a smallest set of atoms should be undefined (i.e., neither true nor false);
- the *principle of justifiability (or foundedness)* You & Yuan (1994): every atom which is true must be derived from the rules of the program, possibly using negative literals as additional axioms.
- the *principle of the closed world assumption (CWA)*, for models of disjunctive logic programs (Eiter *et al.* (1997b)): "If every rule with an atom a in the head

has a false body, or its head contains a true atom distinct from a with respect to an acceptable model, then a must be false in that model."

It can be shown that both the semi-stable and the semi-equilibrium semantics satisfy the first two principles (properly rephrased and viewing **bt** as undefined), but not the CWA principle; this is shown by the simple program $P = \{a \leftarrow not\ a\}$ and the acceptable model $\{Ka\}$. However, this is due to the particular feature of making, as in this example, assumptions about the truth of atoms; if the CWA condition is restricted to atoms a that are not believed by assumption, i.e., $I^K(a) \neq \mathbf{bt}$ in a semi-stable respectively semi-equilibrium model I^K, then the CWA property holds.

We eventually remark that Property **N** can be enforced on semi-stable models by adding constraints $\leftarrow a, not\ a$ for all atoms a to the (original) program. However, enforcing Property **K** on semi-stable models is more involved and efficient encodings seem to require an extended signature.

7.6.2 Related Semantics

P-stable (partial stable) models, which coincide with the 3-valued stable models of Przymusinski (1991a), are one of the best known approximation of answer sets. Recently, the P-stable models have been semantically characterised by Cabalar *et al.* (2007) in the logic HT^2 in terms of partial equilibrium models.

The L-stable models semantics by Eiter *et al.* (1997b), which selects those P-stable models where a smallest set of atoms is undefined, is closest in spirit to semi-stable and semi-equilibrium semantics; furthermore, it satisfies the three principles from above, as well as answer set coverage and congruence (cf. Introduction). The main difference is that L-stable—like P-stable—semantics takes a neutral position on undefinedness, which in combination with negation may lead to weaker conclusions.

For example, the program in Example 7.4.2 has a single P-stable (and L-stable) model in which all atoms are undefined, while c is true under semi-equlibrium semantics. Similarly, the program

$$P = \left\{ \begin{array}{l} a \leftarrow not\ b. \\ b \leftarrow not\ c. \\ c \leftarrow not\ a. \end{array} \right\}$$

has a single P-stable (and thus L-stable) model in which all atoms are undefined, while it has multiple semi-stable models, viz. $\{a, Ka, Kc\}$, $\{b, Kb, Ka\}$, and $\{c, Kc, Kb\}$, which coincide with the semi-equilibrium models. If we add the rules $d \leftarrow a$, $d \leftarrow b$, and $d \leftarrow c$ to P, the new program cautiously entails under both semi-stable and semi-equilibrium model semantics that d is true, but not under L-stable semantics.

Furthermore, disjunctive programs may lack L-stable models, e.g.

$$P' = P \cup \{a \vee b \vee c \leftarrow\};$$

the semi-stable respectively semi-equilibrium models of P' are those of P.

Opposite to the L-stable semantics is the least P-stable model semantics, which selects the P-stable model in which a largest set of atoms is undefined; for normal logic programs, a unique such model always exists, and this model coincides with the well-founded model of van Gelder *et al.* (1991); furthermore, it is characterised in the logic HT^2 in terms of the minimal partial equilibrium model (under a suitable ordering) Cabalar *et al.* (2007). As in the examples for L-stable semantics above, normal programs have a single P-stable model, the least P-stable and the L-stable semantics for these programs coincide, showing thus similar differences to the semi-stable and semi-equilibrium semantics.

Regular semantics You & Yuan (1994) is another 3-valued approximation of answer set semantics that satisfies least undefinedness and foundedness, but not the CWA principle. However, it is classically coherent. For the program P above, the regular models coincide with the L-stable models; the program P' has the regular models $\{a\}$, $\{b\}$, and $\{c\}$. While regular models fulfill answer set coverage, they do not fulfill congruence. For more discussion of 3-valued stable and regular models as well as many other semantics coinciding with them, see Eiter *et al.* (1997b).

Revised stable models Pereira & Pinto (2005) are a 2-valued approximation of answer sets; negated literals are assumed to be maximally true, where assumptions are revised if they would lead to self-incoherence through odd loops or infinite proof chains. For example, the odd-loop program P above has three revised stable models, viz. $\{a,b\}$, $\{a,c\}$, and $\{b,c\}$. The semantics is only defined for normal logic programs, and fulfills answer set coverage but not congruence, cf. Pereira & Pinto (2005). Similarly, pstable models Osorio *et al.* (2008), which have a definition for disjunctive programs however, satisfy answer set coverage (but just for normal programs) and congruence fails. Moreover, every pstable model of a program is a minimal model of the program, but there are programs, such as P above, that have classical models but no pstable models, thus classical coherence does not hold.

7.6.3 Extensions

As already mentioned, semi-stable semantics has originally been developed as an extension to p-minimal model semantics Sakama & Inoue (1995a), a paraconsistent semantics for extended disjunctive logic programs, i.e., programs which besides default negation also allow for strong (classical) negation. A declarative characterisation of p-minimal models by means of frames was given by Alcântara *et al.* (2005), who coined the term *Paraconsistent Answer Set Semantics* (PAS) for it. This characterisation has been further simplified and underpinned with a logical axiomatisation in Odintsov & Pearce (2005) by using Routley models Routley (1974), i.e., a simpler possible world model.

Our characterisations for both, semi-stable models and semi-equilibrium models, can be easily extended to this setting if they are applied to semantic structures which are given by 4-tuples of interpretations rather than bi-interpretations, respectively to Routley here-and-there models rather than HT-models. Intuitively, this again amounts

to considering two 'worlds', each of which consists of a pair of interpretations: one for positive literals (atoms), and one for negative literals (strongly negated atoms). The respective epistemic transformations are unaffected except for the fact that literals are considered rather than atoms. One can also show for both semantics that there is a simple 1-to-1 correspondence to the semi-stable (semi-equilibrium) models of a transformed logic program without strong negation: A given extended program P is translated into a program P' over $\Sigma \cup \{a' \mid a \in \Sigma\}$ without strong negation by replacing each negative literal of the form $\neg a$ by a'. If (I,J) is a semi-stable (semi-equilibrium) model of P', then

$$(I \cap \Sigma, \{\neg a \mid a' \in I\}, J \cap \Sigma, \{\neg a \mid a' \in J\})$$

is a semi-stable (semi-equilibrium) model of P. Note that semi-stable (semi-equilibrium) models of extended logic programs obtained in this way generalise the PAS semantics, which means that they are paraconsistent as well as paracoherent. Logically this amounts to distinguishing nine truth values rather than three, with the additional truth values *undefined*, *believed false*, *believed inconsistent*, *true with contradictory belief*, *false with contradictory belief*, and *inconsistent*. The computational complexity for extended programs is the same.

Compared to Sakama & Inoue (1995a), we further restrict ourselves here to propositional programs, as opposed to programs with variables (non-ground programs). However, the respective semantics for non-ground programs via their groundings are straightforward. Alternatively, in case of semi-equilibrium models one can simply replace HT-models by Herbrand models of quantified equilibrium logic Pearce & Valverde (2008). Similarly for the other semantics, replacing interpretations in the semantic structures by Herbrand interpretations over a given function-free first-order signature, yields a characterisation of the respective models.

7.6.4 Rule Modularity of Semi-Equilibrium Semantics

While the $\mathcal{SEQ}$-semantics has nice properties, it may select models that do not respect a modular structure in the rules. To illustrate this, consider the following example.

Example 7.6.1 *Suppose we have a program that captures knowledge about friends of a person regarding visits to a party, where $go(X)$ informally means that X will go:*

$$P = \left\{ \begin{array}{c} go(John) \leftarrow not\ go(Mark). \\ go(Peter) \leftarrow go(John), not\ go(Bill). \\ go(Bill) \leftarrow go(Peter). \end{array} \right\}$$

Then P has no answer set; its semi-equilibrium models are $M_1 = (\emptyset, \{go(Mark)\})$, and $M_2 = (\{go(John)\}, \{go(John), go(Bill)\})$. Informally, a key difference between M_1 and M_2 concerns the beliefs on Mark and John. In M_2 Mark does not go, and, consequently, John will go (moreover, Bill is believed to go, and Peter will not go). In M_1, instead, we believe Mark will go, thus John will not go (likewise Peter and Bill).

None of the two models provides a fully coherent view (on the other hand, the program is incoherent, having no answer set). Nevertheless, M_2 appears preferable over M_1, since, according with a layering (stratification) principle, which is widely agreed in LP, one should prefer $go(John)$ rather than $go(Mark)$, as there is no way to derive $go(Mark)$ (which does not appear in the head of any rule of the program).

▲

Modularity via rule dependency as in the example above is widely used in problem modeling and logic programs evaluation; in fact, program decomposition is crucial for efficient answer set computation. For the program P above, advanced answer set solvers like DLV and clasp immediately set $go(Mark)$ to false, as $go(Mark)$ does not occur in any rule head. In a customary bottom up computation along program components, answer sets are gradually extended until the whole program is covered, or incoherence is detected at some component (in our example for the last two rules). But rather than to abort the computation, we would like to switch to a paracoherent mode and continue with building semi-equilibrium models, as an approximation of answer sets.

To overcome this limitation, Amendola *et al.* (2014, 2015) introduces a refined paracoherent semantics, called *split semi-equilibrium semantics*. It coincides with the answer sets semantics in case of coherent programs, and selects a subset of the $\mathcal{SEQ}$-models otherwise. Their main results are two model-theoretic characterisations which identify necessary and sufficient conditions for deciding whether a $\mathcal{SEQ}$-model is selected. However, this approach does not attain the property of *compositionality* in the terms we defined previously and as such, further work must be made in that direction.

7.7 Conclusion

We have given a semantic characterisation of semi-stable models in terms of bi-models, and of semi-equilibrium models, which eliminate some anomalies of semi-stable models, in terms of HT-models. Furthermore, we characterised the complexity of major reasoning tasks of these semantics.

Regarding implementation, we developed experimental prototypes for computing $\mathcal{SST}(P)$ and $\mathcal{SEQ}(P)$ based on these characterisations. They construct the bi-models (respectively, HT-models) of P and filter them according to the conditions in Theorem 7.3.1 (respectively, Theorem 7.4.1). Alternatively, $\mathcal{SST}(P)$ and $\mathcal{SEQ}(P)$ are obtainable by postprocessing the answer sets of the epistemic transformation P^K respectively its extension P^{HT}, which are computed with any ASP solver.

Concerning future work, there are several issues. In this Chapter, we have considered paracoherence based on program transformation, as introduced by Sakama & Inoue (1995a). Other notions, like forward chaining construction and strong compatibility Wang *et al.* (2009); Marek *et al.* (1999) might be other candidates to deal with paracoherent reasoning in logic programs, which remain to be explored.

Another subject is to extend paracoherence to language extensions, including aggregates, nested logic programs etc. Of particular interest to us are modular logic programs Janhunen *et al.* (2009); Dao-Tran *et al.* (2009), where module interaction may lead to incoherence. Related to the latter are the more general multi-context systems Brewka & Eiter (2007a), in which knowledge bases exchange beliefs via non-monotonic bridge rules. Based on ideas and results of this Chapter, paracoherent semantics for certain classes of such multi-context systems may be devised.

Another issue is to investigate the use of paracoherent semantics in AI applications such as diagnosis, where assumptions may be exploited to generate candidate diagnoses, in the vein of the generalised stable model semantics Kakas & Mancarella (1990).

Finally, a promising line of work is to apply the $\mathcal{SEQ}$ transformation and check modular compositionality, having in consideration disjunctive MLPs in as much as the $\mathcal{SEQ}$ transformation is also disjunctive.

We provide selected proofs of the results, omitting some details. In particular, we concentrate on the semantic characterisation of semi-stable models in the first part of the chapter. The proofs for semi-equilibrium semantics are similar in spirit.

Chapter 8

Justifications for Answer Set Programs

Viegas Damásio *et al.* (2013) introduced a way to construct propositional formulae encoding provenance information for logic programs. From these formulae, justifications for a given interpretation are extracted but it does not explain why such interpretation is not an answer set (debugging). Resorting to a meta-programming transformation for debugging logic programs, Gebser *et al.* (2008) does the converse.

In this chapter we unify these complementary approaches using meta-programming transformations. First, an answer set program is constructed in order to generate every provenance propositional model for a program, both for well-founded and answer set semantics, suggesting alternative repairs to bring about (or not) a given interpretation. In particular, we identify what changes must be made to a program in order for an interpretation to be an answer set, thus providing the basis to relate provenance with debugging.

With this meta-programming method, one does not have the need to generate the provenance propositional formulas, and thus can obtain debugging and justification models directly from the transformed program. This enables computing provenance answer sets in an easy way by using answer set programming solvers. We show that the provenance approach generalises the debugging one, since any error has a counterpart provenance but not the other way around. Because the method we propose is based on meta-programming, we extended an existing tool (Spock) and developed a proof-of-concept (http://cptkirk.sourceforge.net) to help computing our models.

The most important contributions we make in this chapter are:

(1) bridging the gap between provenance models and logic programs using meta-programming for ASP,

(2) unifying these two complementary approaches by mapping provenance models with debugging models Gebser *et al.* (2008), and

(3) obtainining justifications under the well-founded (WF) and answer set semantics without explicitly calculating provenance formulae for logic programs.

8.1 Introduction and Background

Knowledge was defined by Plato as justified true beliefs but only thousands of years after this philosophical standpoint did Brouwer and the *S*4 provability logic of Gödel formed the intuitionistic view. Since then, the progress towards more justified belief systems has accelerated. Artemov (1995) introduced the Logic of Proofs (LoP) as a formalisation that internalizes justifications for statements and several justification logics were defined based on this LoP, namely Brezhnev (2001). The Knowledge representation and reasoning (KRR) field has also both benefited from and contributed to this direction. Fitting (2005) defined an epistemic semantics for logic programs, Cabalar (2011) used logic programming to define causal logic programming under stable model semantics and, recently, Cabalar *et al.* (2014); Cabalar & Fandinno (2017) presented a multi-valued extension of logic programs associated with a set of justifications expressed in terms of causal graphs which are obtained in a semantic way by algebraic operations and provide an order of rule application.

Theoretical results leading to tools for debugging answer set programs have in the last few years been identified as a crucial prerequisite for a wider acceptance of answer set programming. Tracing approaches, such as Busoniu *et al.* (2013), have been argued not to expose too much the user to the intricacies of reasoners. Because of this, declarative debugging approaches based in meta-programming techniques, such as Brain *et al.* (2007), have instead been developed.

However, in this chapter we are fundamentally interested in addressing the aforementioned questions which we now recall:

Justification models (informally): Answer the question: *Why is a given interpretation indeed an Answer Set?*

Debugging models (informally): Answer the question: *Why is a given interpretation not an Answer Set?*

Example 8.1.1 *Consider* $\Pi = \{\mathbf{r_1} : a \leftarrow b. \quad \mathbf{r_2} : a \leftarrow not\ b.\}$. *Besides knowing a is true in the single AS of* Π*, it is also be important to know that a is true because rules* $\mathbf{r_1}$ *and* $\mathbf{r_2}$ *are in* Π*, independently of what we can conclude about b. One of the other possible justifications for a being true is that* $\mathbf{r_2}$ *is in* Π *and b has no support. Of course, if one intends a to be false then we must conclude that there is a* bug *in the encoding of* Π *as it does not capture our intention, with one justification for it being that rule* $\mathbf{r_2}$ *is unsatisfied.* △

In Viegas Damásio *et al.* (2013), each literal can be associated with its *why not provenance* (WnP), i.e., a logical propositional formula explaining why a literal is true or false in an answer set. In Example 8.1.1, provenance formula

$$Why(a) = (r_1 \wedge \neg not(b)) \vee (r_2 \wedge not(b) \vee \neg not(a))$$

is obtained for literal a and its negation for *not* a. Clearly, $r_1 \wedge r_2 \models Why(a)$ is an intuitive justification for a.

This provenance approach is capable of providing the corrections (adding or removing facts, and removing rules, e.g., $not(a) \wedge not(b) \wedge \neg r_2 \models \neg Why(a)$ in Example 8.1.1) that are sufficient to bring about certain intended models, indicating that the removal of rule r_2 from the program, and neither adding facts for a nor b, makes a false.

Provenance formulas are inspired by a program transformation previously defined for declarative debugging of logic programs Pereira *et al.* (1993b) and have been conjectured in Viegas Damásio *et al.* (2013) to be related with debugging.

The debugging of ASP programs has been addressed in the literature by several authors, and the most effective approaches resort to meta-transformations to detect the diverse forms of anomalies in programs Brain *et al.* (2007); Gebser *et al.* (2008); Oetsch *et al.* (2010); Polleres *et al.* (2013), which, being very fine grained, are designed to pinpoint errors in logic programs. On a different stance, Eiter *et al.* (2010a) provided two approaches for explaining inconsistencies, both of which characterise inconsistency in terms of bridge rules: by pointing out rules which need to be altered for restoring consistency, and by finding combinations of rules which cause inconsistency.

Next, in the remainder of this section, we review relevant logic programming formalisms followed by debugging and provenance literature. In Section 8.2 we introduce a novel meta-programming transformation both for well-founded and answer set semantics that is used to obtain models for WnP formulas of a given normal logic program. We clarify which models are justifications, define them in terms of answer set existence for the meta-program and discuss computational complexity. Section 8.4 provides a new mapping between our and the debugging transformations, showing that provenance captures debugging models but not the other way around. We end with a discussion, a comparison of these approaches with others in the literature and possible future work.

8.1.1 Debugging of Answer Set Programs

Debugging of logic programs and in particular ASP has received important contributions over the last years.

We are mostly interested in Gebser *et al.* (2008) though, where a meta-programming technique for debugging ASPs is presented. Debugging queries are expressed by means of ASP programs, which allows restricting debugging information to relevant parts. The main question addressed is: "Why are interpretations that we expected to be answer sets, indeed not answer sets of a given ASP program?" Thus it finds semantic errors in programs. The provided explanations are based on a scheme of errors relying in a characterisation of answer set semantics by Lee (2005); Ferraris *et al.* (2007). In Theorem 2 of Gebser *et al.* (2008), the four possible causes of errors dealt by their debugging framework are:

Unsatisfied rules: If rule $r \in Gr(\Pi)$, with nonempty $Head(r)$, is unsatisfied by an interpretation I, the logical implication expressed by r is false with respect to I, i.e., $I \models Body(r)$ and $Head(r) \notin I$, and thus I is not a classical model of Π.

Violated integrity constraints: If an integrity constraint (IC) $c \in Gr(\Pi)$ is applicable with respect to an interpretation I, the fact that $Head(c) = \emptyset$ implies $I \not\models c$, and thus I cannot be an answer set of Π. This can be seen as a particular case of unsatisfied rule.

Unsupported atoms: If $\{a\} \subseteq I$ is unsupported with respect to an interpretation I, no rule in $Gr(\Pi)$ allows for deriving a, and thus I is not a minimal model of Π^I.

Unfounded loops: If a loop $\Gamma \subseteq I$ of Π is unfounded with respect to an interpretation I, there is no acyclic derivation for the atoms in Π, and thus I is not a minimal model of Π^I.

The modules in Figures 8.1, 8.2, and 8.3 capture exactly these four causes of errors for disjunctive logic programs. Note here that $head/2$ (respectively $body/2$) is used to relate a rule identifier with its head (respectively with its body), $atom/1$ capture all possible atoms in the vocabulary of a program and $int/1$ captures the interpretation we are evaluating ($int(A)$ means that atom A is true whereas $\overline{int}(A)$ means that it is false). The program π_{int} is a choice loop producing every possible interpretation. The program π_{sat} determines if rules are unsatisfied, when they are applicable but the head is not in that particular interpretation, and if an integrity constraint is violated. The program π_{supp} determines if an atom is unsupported, whereas π_{ufloop} detects unfounded loops.

Finally, in π_{noas}, an interpretation is marked as not being an answer set if it contains one of the error indicating predicates.

$$\begin{aligned}
\pi_{int} = \{ & int(A) \leftarrow atom(A), not\ \overline{int}(A). \\
& \overline{int}(A) \leftarrow atom(A), not\ int(A).\}
\end{aligned}$$

$$\begin{aligned}
\pi_{sat} = \{ & hasHead(R) \leftarrow head(R,_). \\
& someHInI(R) \leftarrow head(R,A), int(A). \\
& violated(C) \leftarrow ap(C), not\ hasHead(C), hasHead(R). \\
& unsatisfied(R) \leftarrow ap(R), not\ someHInI(R).\}
\end{aligned}$$

$$\begin{aligned}
\pi_{supp} = \{ & unsupported(A) \leftarrow int(A), not\ supported(A). \\
& supported(A) \leftarrow head(R,A), ap(R), not\ otherHInI(R,A). \\
& otherHInI(R,A1) \leftarrow head(R,A2), int(A2), head(R,A1), A1 \neq A2.\}
\end{aligned}$$

Figure 8.1: Static Modules of Meta-Program $D(\Pi)$ (Part 1).

$$
\begin{aligned}
\pi_{noas} = \{ & noAnswerSet \leftarrow unsatisfied(\_). \\
& noAnswerSet \leftarrow violated(\_). \\
& noAnswerSet \leftarrow unsupported(\_). \\
& noAnswerSet \leftarrow ufLoop(\_). \\
& \leftarrow not\ noAnswerSet. \}
\end{aligned}
$$

$$
\begin{aligned}
\pi_{ap} = \{ & bl(R) \leftarrow Body^{+}(R,A), \overline{int}(A). && \%blocked\ rules \\
& bl(R) \leftarrow Body^{-}(R,A), int(A). \\
& ap(R) \leftarrow rule(R), not\ bl(R). \} && \%applicable\ rules
\end{aligned}
$$

Figure 8.2: Static Modules of Meta-Program $D(\Pi)$ (Part 2).

$$
\begin{aligned}
\pi_{ufloop} = \{ & ufLoop(A) \leftarrow int(A), supported(A), not\ \overline{ufloop}(A). \\
& \overline{ufloop}(A) \leftarrow int(A), not\ ufLoop(A). \\
& someBInLoop(R) \leftarrow Body^{+}(R,A), ufLoop(A). \\
& someHOutLoop(R) \leftarrow head(R,A), \overline{ufloop}(A). \\
& dpcy(A1,A2) \leftarrow dpcy(A1,A3), dpcy(A3,A2). \\
& dpcy(A1,A2) \leftarrow head(R,A1), Body^{+}(R,A2), ap(R), ufLoop(A1), \\
& \qquad ufLoop(A2), not\ someHOutLoop(R). \\
& \leftarrow head(R,A), ufLoop(A), ap(R), \\
& \qquad not\ someHOutLoop(R), not\ someBInLoop(R). \\
& \leftarrow ufLoop(A1), ufLoop(A2), not\ dpcy(A1,A2). \}
\end{aligned}
$$

Figure 8.3: Static Modules of Meta-Program $D(\Pi)$ (Part 3).

Gebser *et al.* (2008) construct a meta-program from a given program Π and interpretation I that is capable of detecting the errors we enumerated above via occurrences of the following **error-indicating meta-atoms** (or error indicating predicates) in its answer sets: $unsatisfied(l_r)$ indicates that a rule $r \in Gr(\Pi)$ is unsatisfied by an interpretation I; $violated(l_c)$ indicates that an integrity constraint $c \in Gr(\Pi)$ is violated with respect to an interpretation I; $unsupported(l_a)$ indicates that an atom $a \in I$ is unsupported; and $ufLoop(l_a)$ indicates that an atom a belongs to some unfounded loop $\Gamma \subseteq I$ of Π with respect to an interpretation I. Note here that l_r, l_c, and l_a are literals representing respectively a rule, an integrity constraint, and an atom.

Still in Gebser *et al.* (2008), the authors define the input meta-program $\pi_{in}(\Pi)$ from a ground disjunctive logic program Π (note that later on we restrict our discussion to normal logic programs) as the set of facts we depict in Figure 8.4:

Definition 8.1.1 (Module $\pi_{in}(\Pi)$) *Program module $\pi_{in}(\Pi)$ consists of facts stating which rules and atoms occur in Π and, for each rule $r \in \Pi$, which atoms are con-*

$$\begin{aligned}\pi_{in}(\Pi) = \{atom(l_a) \leftarrow \mid a \in At(\Pi)\} \cup \\ \{rule(l_r) \leftarrow \mid r \in \Pi\} \cup \\ \{Head(l_r, l_a) \leftarrow \mid r \in \Pi, a \in Head(r)\} \cup \\ \{Body^+(l_r, l_a) \leftarrow \mid r \in \Pi, a \in Body^+(r)\} \cup \\ \{Body^-(l_r, l_a) \leftarrow \mid r \in \Pi, a \in Body^-(r)\}\end{aligned}$$

Figure 8.4: Module $\pi_{in}(\Pi)$.

tained in its head ($Head(r)$), and body ($Body^+(r)$ and $Body^-(r)$), and is depicted in Figure 8.4. ▲

Given $\pi_{in}(\Pi)$, the non-disjunctive **meta-program** $D(\Pi)$ is defined as follows:

Definition 8.1.2 (Meta Program $D(\Pi)$) *Let Π be a ground DLP. Then, the meta-program $D(\Pi)$ for Π consists of $\pi_{in}(\Pi)$ together with the modules of Figures 8.1, 8.2, and 8.3 , i.e., $D(\Pi) = \pi_{in}(\Pi) \cup \pi_{int} \cup \pi_{ap} \cup \pi_{sat} \cup \pi_{supp} \cup \pi_{ufloop} \cup \pi_{noas}$.* ▲

Example 8.1.2 *Consider program $P = \{\mathbf{r_1} : a \leftarrow b, c.\ \mathbf{r_2} : b \leftarrow d.\ \mathbf{r_3} : b \leftarrow not\ e.\ \mathbf{f_1} : c.\ \mathbf{f_2} : d.\}$, for which an intended AS is $I = \{b, c, d, e\}$. An explanation for I not being an AS is that r_1 is unsatisfied and e is unsupported. On the other hand, Gebser et al. (2008) cannot say why $\{a, b, c, d\}$ is an answer set because a is true due to d being true, and to e being false.* △

Example 8.1.3 *Consider now a program with a positive loop and an integrity constraint: $\Pi_2 = \{r_1 :\ a \leftarrow b.\ \ r_2 :\ b \leftarrow a.\ \ ic : \leftarrow a, b.\}$ where both a and b are expected to be unsupported. The evaluation of $D(\Pi_2)$ yields 4 models with desirable debugging explanations, from which, one is: $\{violated(ic), \sim ufLoop(a), \sim ufLoop(b), supported(a), supported(b), noAnswerSet\}$ meaning that the ic is violated when the atoms have support outside the loop. This model is predicted in Theorem 4 of Gebser et al. (2008).*

△

As we have shown in Example 8.1.2, these approaches do not answer the question of why a given (possibly unintended) interpretation is indeed an AS.

8.1.2 Provenance

In turn, Viegas Damásio *et al.* (2013) present a declarative approach to extract why not provenance information for logic programs. Using values of a freely generated Boolean algebra as annotation tags for atoms, they specify WnP for positive and normal logic programs under the well-founded semantics, and relate it to abduction and calculation of prime implicants. The authors then propose a generalisation to ASP. These WnP formulae are used to determine provenance of literals true in a given

model, and are shown in Viegas Damásio *et al.* (2013) to extend the approaches of evidence graphs Pemmasani *et al.* (2004) and off-line justifications Pontelli *et al.* (2009). In the remaining of this section, assume that a logic program Π over Herbrand base H_Π is given.

Why-not provenance (WnP) is defined in Viegas Damásio *et al.* (2013) and summarised below:

Definition 8.1.3 *Let B_Π be the free Boolean algebra generated by propositional variables of the following form:*

$$H_\Pi \cup not(H_\Pi) \cup \{r_i \mid 1 \leq i \leq |\Pi|\}$$

where for each rule of Π there is a unique and distinct rule identifier r_i. Elements of B_Π are the equivalence classes of propositional formulas under logical equivalence, and the partial ordering of B_Π is entailment:

$$[\phi] \preceq [\psi] \text{ iff } \phi \models \psi$$

Thus B_Π is a lattice, with join and meet defined respectively by

$$[\phi] \oplus [\psi] = [\phi \vee \psi]$$

$$[\phi] \otimes [\psi] = [\phi \wedge \psi]$$

and let

$$[\phi] - [\psi] = [\phi \wedge \neg\psi]$$

▲

WnP is extracted with monotonic multivalued programs and a **WnP program $\mathfrak{P}$ over** H_Π is defined as:

Definition 8.1.4 *Let a WnP program $\mathfrak{P}$ be formed by rules of the form*

$$A \Leftarrow [J] \otimes B_1 \otimes \ldots \otimes B_m \text{ with } m \geq 0$$

$$\text{and where } [J] \in B_\Pi \text{ and } A, B_1, \ldots, B_m \in H_\Pi$$

An interpretation I for $\mathfrak{P}$ is a mapping $I : H_\Pi \to B_\Pi$. The set of all interpretations is a lattice with point-wise ordering.

An interpretation I satisfies a rule

$$A \Leftarrow [J] \otimes B_1 \otimes \ldots \otimes B_m$$

of program $\mathfrak{P}$ iff

$$I(A) \succeq [J] \otimes I(B_1) \otimes \ldots \otimes I(B_m)$$

iff

$$J \wedge I(B_1) \wedge \ldots \wedge I(B_m) \models I(A)$$

Interpretation I is a model of $\mathfrak{P}$ iff I satisfies all the rules of $\mathfrak{P}$. ▲

The monotonicity of all rules of $\mathfrak{P}$ guarantees the existence of a least model $M_{\mathfrak{P}}$ for it, and by mimicking the construction of a Gelfond-Lifschitz like operator, why not provenance for logic programs under the well-founded semantics can be defined.

Definition 8.1.5 (Provenance program $\frac{\mathfrak{P}}{\mathbf{I}}$) *is constructed from* Π *and WnP interpretation I as follows:*

- *For the* i^{th} *rule* $A \leftarrow B_1,\ldots,B_m,not\ C_1,\ldots,not\ C_n$ $(m+n \geq 1)$ *in* Π *add provenance rule*
$$A \Leftarrow [r_i \wedge \neg I(C_1) \wedge \ldots \neg I(C_n)] \otimes B_1 \otimes \ldots \otimes B_m$$
to $\frac{\mathfrak{P}}{I}$*;*
- $\forall A \in H_\Pi$*, if* $A \in \Pi$ *(respectively,* $A \notin \Pi$*), add the following to* $\frac{\mathfrak{P}}{I}$*:*
$$A \Leftarrow [A] \text{ (respectively, } A \Leftarrow [\neg not(A)])$$

Operator $\mathfrak{G}_\Pi(I) = M_{\frac{\mathfrak{P}}{I}}$ *returns the least model of* $\frac{\mathfrak{P}}{I}$*.*
▲

The **rationale** for $\mathfrak{P}$ is: If a program $\mathfrak{P}$ has some fact A (respectively, no fact for A), WnP formula for A is

$$[(r_i \wedge Why_i) \vee \ldots \vee (r_j \wedge Why_j) \vee A]$$

$$\text{(respectively, } [(r_i \wedge Why_i) \vee \ldots \vee (r_j \wedge Why_j) \vee \neg not(A)]).$$

A justification for A is $[A]$ meaning there is a fact for A. Other justifications are obtained using a rule r_k and justifying why its body is true. The later case (denoted before by 'respectively') is better understood taking the justification for *not A* which has WnP formula $[\neg(r_i \wedge Why_i) \wedge \ldots \wedge \neg(r_j \wedge Why_j) \wedge not(A)]$, expressing that all bodies must be falsified and $[not(A)]$ holds.

Example 8.1.4 (Provenance program) *Consider again the program in Example 8.1.2:* $P = \{\mathbf{r_1} : a \leftarrow b, c.\ \ \mathbf{r_2} : b \leftarrow d.\ \ \mathbf{r_3} : b \leftarrow not\ e.\ \ \mathbf{f_1} : c.\ \ \mathbf{f_2} : d.\}$*, for which an intended AS is* $I = \{a : false, b : false, c : false, d : false, e : false\}$*. Note that in the second rule for b, corresponding to* r_3*,* $\neg I(e)$ *originates true.*

Its corresponding provenance program is:

$$\mathfrak{P} = \left\{ \begin{array}{lll} a & \Leftarrow & [r1] \otimes b \otimes c. \\ a & \Leftarrow & [\neg not\ a]. \\ b & \Leftarrow & [r2] \otimes d. \\ b & \Leftarrow & [r3 \wedge true]. \\ b & \Leftarrow & [\neg not\ b]. \\ c & \Leftarrow & [c]. \\ d & \Leftarrow & [d]. \end{array} \right\}$$

△

Operator $\mathfrak{G}_\Pi$ is anti-monotonic and therefore $\mathfrak{G}_\Pi^2$ is monotonic having a least model $\mathfrak{T}_\Pi$, corresponding to provenance information for what is true in the WFM, while $\mathfrak{TU}_\Pi = \mathfrak{G}_\Pi(\mathfrak{T}_\Pi)$ contains the WnP of what is true or undefined in the WFM of Π. The following definitions capturing **Why-not provenance information under the well-founded semantics** are then provided:

Definition 8.1.6 *Let $\mathfrak{T}_\Pi$ be the least model of $\mathfrak{G}_\Pi^2$, $\mathfrak{TU}_\Pi = \mathfrak{G}_\Pi(\mathfrak{T}_\Pi)$, and A be an atom. Let:*

$$\begin{aligned} Why_\Pi(A) &= [\mathfrak{T}_\Pi(A)] \\ Why_\Pi(not\ A) &= [\neg\mathfrak{TU}_\Pi(A)] \\ Why_\Pi(undef\ A) &= [\neg\mathfrak{T}_\Pi(A) \wedge \mathfrak{TU}_\Pi(A)] \end{aligned}$$

▲

Theorem 8.1.1 (Provenance Models for the Well-Founded Semantics) *Let $G \notin \Pi$ and $F \in \Pi$ (respectively, $R \in \Pi$) be sets of facts (respectively, rules). Literal $L \in WFM((\Pi \setminus (F \cup R)) \cup G)$ iff there is a conjunction of literals $C \models Why_\Pi(L)$ such that*

$$\begin{aligned} &\mathbf{removeFact}(\mathbf{C}) \subseteq F, \\ &\mathbf{keepFact}(\mathbf{C}) \cap F = \emptyset, \\ &\mathbf{removeRule}(\mathbf{C}) \subseteq R, \\ &\mathbf{keepRule}(\mathbf{C}) \cap R = \emptyset, \\ &\mathbf{missingFact}(\mathbf{C}) \subseteq G, \text{ and} \\ &\mathbf{noFact}(\mathbf{C}) \cap G = \emptyset \end{aligned}$$

where $noFact(C)$ (respectively, $missingFact(C)$) are facts that cannot (respectively, must) be added:

$$\begin{aligned} &\mathbf{keepFact}(\mathbf{C}) = \{A. \mid A \in F\} \\ &\mathbf{removeFact}(\mathbf{C}) = \{A. \mid \neg A \in F\} \\ &\mathbf{keepRule}(\mathbf{C}) = \{r_i : A \leftarrow Body \mid r_i \in R \text{ and } r_i \in \Pi\} \\ &\mathbf{removeRule}(\mathbf{C}) = \{r_i : A \leftarrow Body \mid \neg r_i \in R \text{ and } r_i \in \Pi\} \\ &\mathbf{noFact}(\mathbf{C}) = \{A. \mid not(A) \in G\} \\ &\mathbf{missingFact}(\mathbf{C}) = \{A. \mid \neg not(A) \in G\} \end{aligned}$$

○

We henceforth call **repair indicating predicates** (or simply **repair predicates**) to: removeFact/1, keepFact/1, removeRule/1, keepRule/1, missingFact/1, noFact/1

Example 8.1.5 (From Viegas Damásio *et al.* (2013)) *Consider again program Π in*

Example 8.1.2. Its WnP information is:

$$\begin{aligned}
Why(a) &= [r_1 \wedge c \wedge ((r_2 \wedge d) \vee (r_3 \wedge not(e)) \vee \neg not(b)) \vee \neg not(a)] \\
Why(not\ a) &= [not(a) \wedge (\neg r_1 \vee \neg c \vee (not(b) \wedge \\
&\qquad (\neg r_2 \vee \neg d) \wedge (\neg r_3 \vee \neg not(e)))]
\end{aligned}$$

$$\begin{aligned}
Why(b) &= [(r_2 \wedge d) \vee (r_3 \wedge not(e)) \vee \neg not(b)] \\
Why(not\ b) &= [not(b) \wedge (\neg r_2 \vee \neg d) \wedge (\neg r_3 \vee \neg not(e))]
\end{aligned}$$

$$\begin{aligned}
Why(c) &= [c] \\
Why(not\ c) &= [\neg c]
\end{aligned}$$

$$\begin{aligned}
Why(d) &= [d] \\
Why(not\ d) &= [\neg d]
\end{aligned}$$

$$\begin{aligned}
Why(e) &= [\neg not(e)] \\
Why(not\ e) &= [not(e)]
\end{aligned}$$

The provenance for atom a taking the truth value false, and all other atoms taking the value true is derived from the models of the following conjunction of provenance formulae:

$$Why(not\ a) \wedge Why(b) \wedge Why(c) \wedge Why(d) \wedge Why(e) =$$
$$= not(a) \wedge \neg r_1 \wedge (r_2 \vee \neg not(b)) \wedge c \wedge d \wedge \neg not(e)$$

Thus, translating this to our program repair predicates, we get that a fact for a must be absent ($noFact(a)$), we have to remove rule r_1, keep rule r_2 or add fact b ($missingFact(b)$), keep rules encoding facts for c and d, and add fact e.

Gebser et al. *(2008) detect that rule r_1 is unsatisfied and e is unsupported but their approach does not determine provenance for a.*

One way to make a true is to simply add a fact for it; alternatively r_1 must be kept in Π as well as facts c and b. This is achieved by keeping r_2 and d, keeping r_3 and not adding e, or adding b. ■

Still from Viegas Damásio *et al.* (2013), one obtains AS provenance from the WFM provence as follows:

Definition 8.1.7 (Provenance for Answer Set Semantics) *Let Π be a logic program, and L a literal. The answer set why-not provenance for literal L is*

$$AnsWhy_{\Pi}(L) = Why_{\Pi}(L) \wedge \bigwedge_{A \in H_{\Pi}} \neg Why_{\Pi}(undefA)$$

▲

This is done basically by forcing all atoms to be either positive or negative, i.e., non-undefined, and using the provenance determined for the well-founded semantics (see examples in Section 8.2). **Justifications** are defined as:

Definition 8.1.8 (Justifications for Well-Founded Semantics) *Given a logic program* Π *and a literal* l*, a justification* J *for* l *in* Π *as an implicant of the why provenance formula* $Why_{\Pi}(l)$ *, i.e., a conjunction of literals such that* $J \models Why_{\Pi}(l)$*. The implicant is* prime *if there is no other implicant* J' *of* $Why_{\Pi}(l)$ *with less literals.* ▲

Note that it has been proved in Viegas Damásio *et al.* (2013) that evidence graphs Pemmasani *et al.* (2004) and off-line justifications Pontelli *et al.* (2009) models can all be captured by WnP implicants, but some of our justifications cannot be mapped by them.

Also related to provenance are causal chains Cabalar (2011); Cabalar & Fandiño (2013); Cabalar *et al.* (2014); Cabalar & Fandinno (2017) where a multi-valued semantics for normal logic programs whose truth values form a lattice of causal chains is provided. A causal chain is a concatenation of rule labels reflecting some sequence of rule applications. In Cabalar & Fandinno (2017), the authors have shown that the existence of enabled justifications is a sufficient and necessary condition for the truth of a literal. Furthermore, their causal justifications capture, under the well-founded semantics, both Causal Graphs and Why-not Provenance justifications. They have also established a formal relation between these two approaches under this semantics.

They also proved a direct correspondence between the semantic values they obtain and the syntactic idea of proof and extrapolated those results to stable models of programs with default negation, understanding the latter as "absence of cause".

Theorem 8.1.2 (In Viegas Damásio *et al.* (2013)) *Let* P *be a program,* M *an answer set of* P*, and* L *a literal true in* M*. Then there is a conjunction* $C \models AnsWhy_P(L)$ *such that* C *does not contain any negative literals for literals true in the* $WFM(P)$*. For every atom* $A \in M$ *but which is undefined in* $WFM(P)$*,* C *includes* $not(A) \wedge \neg r_{i_1} \wedge \ldots \wedge \neg r_{i_k}$ *where* $\{r_{i_1}, \ldots, r_{i_k}\}$ *is the set of identifiers of all rules for* A*.* ○

The above theorem follows from the result in Pontelli & Son (2006) stating that there is an offline justification with respect to M and the set of assumptions containing the literals false in M that are undefined in the well-founded model of P. Moreover, this justification does not have cycles. Therefore, by representing these assumptions as a conjunction of literals, we can then construct such a C. In order to assume a literal false we cannot add a fact for it ($not(A)$), and must remove all existing rules for it in the program ($\neg r_{i_1} \wedge \ldots \wedge \neg r_{i_k}$).

8.2 Provenance Transformation for the Well-Founded Semantics

We define here a novel program transformation, capable of obtaining all models of WnP formulae, composed of two parts:

1. a set of common modules in Fig. 8.5, shared by specific transformations for both the well-founded and the answer set semantics;

2. Modules specific to the well-founded semantics in Fig. 8.6.

Its vocabulary is based in Gebser *et al.* (2008) to ease their subsequent combination. We only deal with the non-disjunctive case and ICs must be represented in their explicit form (i.e., $ic \leftarrow Body, not\ ic.$).

$$\begin{aligned}\pi_{fact} = \{&fact(X) \leftarrow rule(R), head(R,X), not\ hasBody(r).\\ &hasBody(r) \leftarrow rule(R), bodyP(R,A).\\ &hasBody(r) \leftarrow rule(R), bodyN(R,A).\}\end{aligned}$$

$$\begin{aligned}\pi_{Rules} = \{&keepRule(R) \leftarrow rule(R), not\ removeRule(R).\\ &removeRule(R) \leftarrow rule(R), not\ keepRule(R).\}\end{aligned}$$

$$\begin{aligned}\pi_{Facts} = \{&missingFact(X) \leftarrow atom(X), not\ fact(X), not\ noFact(X).\\ &noFact(X) \leftarrow atom(X), not\ fact(X), not\ missingFact(X).\}\end{aligned}$$

Figure 8.5: Common Provenance Modules $\pi_{common} \cup \pi_{fact} \cup \pi_{Rules} \cup \pi_{Facts}$

Module π_{fact} defines facts as rules in the program having empty bodies. Module π_{fact} assumes that module π_{in} of $D(\Pi)$ (Fig. 8.4) will also be applied, since it depends on all facts defined in π_{in}. Modules π_{Rules} and π_{Facts} are the generators for propositional variables used in the provenance formulae in the vocabulary of Theorem 8.1.1. Note that in π_{Rules} the provenance propositional variables for facts H_Π are captured by $keepRule/1$ since, for generality purposes, $rule/1$ represents both rule and fact identifiers.

8.2.1 Provenance for the Well-Founded Semantics

A provenance program under the well-founded Semantics is captured by π_{wfs} combined with debugging modules π_{common} and π_{in}. Module π_{ltu} encodes the Γ^2 operator for the program subject to changes defined by pairs $keepRule/removeRule$ and $missingFact/noFact$, where predicate $t/1$ represents what is true (the outer Γ), and $tu/1$ what is true or undefined (the inner Γ). The constraint discards models where assignments are contradictory, ensuring that $t(A) \Rightarrow tu(A)$ for every atom A. The module also uses an extra meta-predicate $undef/1$ that allows to make an atom undefined, a new kind of change not captured by the original provenance model for well-founded semantics that is included for the sake of completeness. Module π_{apttu} determines when a rule is applicable in the outer $(ap(R,t))$, and inner steps $(ap(R,tu))$, and generalises module π_{ap} of Gebser *et al.* (2008).

We use W as an abbreviation for the provenance WF transformation produced by all relevant modules:

$$W(\Pi) = \pi_{in}(\Pi) \cup \pi_{common} \cup \pi_{wfs}(\Pi)$$

$$
\begin{aligned}
\pi_{ttu} = \{\ & \leftarrow atom(A), t(A), not\ tu(A). \\
& t(H) \leftarrow head(R,H), keepRule(R), ap(R,t), not\ undef(H). \\
& t(H) \leftarrow atom(H), missingFact(H), not\ fact(H),\ not\ undef(H). \\
& tu(H) \leftarrow head(R,H), keepRule(R), ap(R,tu). \\
& tu(H) \leftarrow atom(H), missingFact(H), not\ fact(H). \\
& tu(H) \leftarrow atom(H), undef(H).\}
\end{aligned}
$$

$$
\begin{aligned}
\pi_{apttu}(\Pi) = \{ & ap(r_i,t) \leftarrow t(B_1), \ldots t(B_m), not\ tu(C_1), \ldots, not\ tu(C_n)., \\
& ap(r_i,tu) \leftarrow tu(B_1), \ldots tu(B_m), not\ t(C_1), \ldots, not\ t(C_n). \\
& \quad \mid A \leftarrow B_1, \ldots B_m, not\ C_1, \ldots, not\ C_n \in \Pi \\
& \qquad \text{is identified by } r_i.\}
\end{aligned}
$$

Figure 8.6: Meta transformation π_{wfs} modules

We define a reparation formula and a repaired program for $W(\Pi)$ in the following way:

Definition 8.2.1 (Reparation Formula) *Let Π be a program and $M \in AS(W(\Pi))$ be an answer set of its provenance WF transformation. A reparation formula $Rep(M)$ is conjunction of repair predicates R in the vocabulary of Theorem 8.1.1, namely: removeRule; keepRule; missingFact or noFact, s.t. $R \in M$.* ▲

Definition 8.2.2 (Repaired Program) *Let Π be a program and $M' \in AS(W(\Pi))$ be an answer set of its provenance WF transformation. We construct from M' a repaired program Π' by deleting from Π every rule identified by r_i such that $removeRule(r_i) \in M'$, and adding a fact A to Π' for every $missingFact(r_{A\leftarrow}) \in M'$.* ▲

Lemma 15 (Reverse Provenance Models) *Let Π be a program, $M' \in AS(W(\Pi))$ be an answer set of its provenance WF transformation, and*

$$Model(M') = \{A \mid t(A) \in M'\} \cup \{not\ A \mid tu(A) \notin M'\}.$$

Let Π' be a repaired program, constructed as in Definition 8.2.2. Then, $Model(M')$ is a PSM of Π'. Conversely, if Π' is a program obtained by deleting rules or adding facts, then every PSM of Π' has a corresponding AS in $W(\Pi)$. ○

Proof of Lemma 15.

(i) Module π_{ttu} encodes the Γ^2 operator for the program subject to changes defined by pairs *keepRule*/*removeRule* and *missingFact*/*noFact*, where predicate $t/1$ represents what is true (the outer Γ), and $tu/1$ what is true or undefined (the inner Γ).

(ii) The constraint in π_{ttu} discards models where assignments are contradictory, ensuring that $t(A) \Rightarrow tu(A)$ for every atom A.

Note also that module π_{apttu} determines when a rule is applicable in the outer $(ap(R,t))$, and inner steps $(ap(R,tu))$.

(←) Assume there is an interpretation $M' = \Gamma_{W(\Pi)}(M')$. Then by the definition of an answer set, $M' \in AS(W(\Pi))$. If we do the reduct $W(\Pi)^{M'}$ we obtain the repaired program divided by the set of T atoms with respect to M' (the set $\{A \mid t(A) \in M'\}$). The set of TU atoms with respect to M' corresponds to the rules that are thus left in the repaired program reduct with respect to to M'. Because of (ii), TU atoms have a bi-univocal correspondence to atoms is $\Gamma(T)$ and if we inject the TU atoms in the rules for T and inject the T atoms in the rules that are applicable for $\Gamma(TU)$, because we are applying the operator to a fixed point and obtain nothing new, this fixed point is a model and is a PSM.

(→) Assume that there is a partial stable model $P_{sm} \in PSM(\Pi')$ which does not have a corresponding AS because either it is not a model, of because there is a smaller PSM $P'_{sm} \in PSM(\Pi')$ which is indeed a model. That implies that T/TU literals of the smaller model P'_{sm} would be contained in P_{sm} which implies that, because by (ii) a comparable model cannot differ only on TU atoms but rather also on the corresponding T atoms, and because we know by (i) that T rules mimic the outer Γ^2 operator which corresponds to the answer sets and TU rules mimic the inner Γ operator and correspond to every partial stable model, if P_{sm} is not a partial stable model then it is not a model of Γ then, it is also not a model of the outer Γ^2 operator and thus not an answer set of $W(\Pi)$ which contradicts our assumption. □

Lemma 16 *Given a logic program Π and a propositional model M of WnP formula for a literal L, namely $Why_\Pi(L)$, then there is an AS M' of $W(\Pi) = \pi_{in}(\Pi) \cup \pi_{common} \cup \pi_{wfs}(\Pi)$ s.t. $L \in Model(M')$ and:*

1. *If $A \in H_\Pi$ such that fact $A \in \Pi$ and $A \in M$, then $keepRule(r_{A\leftarrow}) \in M'$ and $removeRule(r_{A\leftarrow}) \notin M'$.*
2. *If $A \in H_\Pi$ such that fact $A \in \Pi$ and $A \notin M$, then $keepRule(r_{A\leftarrow}) \notin M'$ and $removeRule(r_{A\leftarrow}) \in M'$.*
3. *If $not(A) \in H_\Pi$ such that no fact A does occur as a fact in Π and $not(A) \notin M$, then $missingFact(A) \in M'$ and $noFact(A) \notin M'$.*
4. *If $not(A) \in H_\Pi$ such that no fact A does occur as a fact in Π and $not(A) \in M$, then $missingFact(A) \notin M'$ and $noFact(A) \in M'$.*
5. *If $r_i \in M$, then $keepRule(r_i) \in M'$ and $removeRule(r_i) \notin M'$.*
6. *If $r_i \notin M$, then $keepRule(r_i) \notin M'$ and $removeRule(r_i) \in M'$.*

○

Proof of Lemma 16. Let M be a model of the provenance formula $Why_\Pi(L)$, then there is an answer set $M' \in AS(W(\Pi))$ such that $L \in Model(M')$, hence

$M \models Why_{\Pi}(L)$. Then, it is possible to construct a repaired program by removing facts and rules such that $\neg r_i \in M$ and a $\neg A \in M$, respectively, and add facts such that $\neg not(A) \in M$. This originates a repaired program $\Pi \backslash (F \cup R) \cup G$ and we know from theorem 8.1.1 that $L \in WFM(\Pi \backslash (F \cup R) \cup G)$. By Lemma 15, every PSM of a repaired program belongs to $AS(W(\Pi))$.

The rest of the proof follows by construction of the repair predicates:

1. Because $fact(A)$ is in the program, the only applicable rules making $t(A)$ and $tu(A)$ true and thus making A true in the model, would imply that $keepRule(A)$ must be in the model and, because they are mutually exclusive, $removeRule(r_{A\leftarrow}) \notin M'$.

2. Because $fact(A)$ is in the program, the only way to make $t(A)$ and $tu(A)$ false and thus to make A false in the model, is to not have $keepRule(r_{A\leftarrow})$ in the model and, because of the choice loop between keepRule and removeRule predicates, $removeRule(r_{A\leftarrow}) \in M'$.

3. Because $fact(A)$ is not in the program, the only applicable rules making $t(A)$ and $tu(A)$ true and thus making A true in the model, would imply that $missingFact(A)$ must be in the model and, because they are mutually exclusive, $noFact(A) \notin M'$.

4. Because $fact(A)$ is not in the program, the only way to make $t(A)$ and $tu(A)$ false and thus to make A false in the model, is to not have $missingFact(A)$ in the model and, because of the choice loop between missingFact and noFact predicates, $noFact(r_A) \in M'$.

5. The same as item 1, where a fact is taken as a particular case of a rule.

6. The same as item 2, where a fact is taken as a particular case of a rule.

For the converse direction, extra answer sets of $W(\Pi)$ may be generated. When we fix the changes to Π all partial stable models (PSM) of the changed program are obtained, i.e., all fixed points of Γ^2, instead of solely the least fixed point of Γ^2. These models can be filtered out by guaranteeing minimality of the model.

Lemma 17 *Let M' be an AS of $W(\Pi)$, such that there is no $M'' \in AS(W(\Pi))$ for which $Model(M'') \subset Model(M')$ and s.t. coincide on the truth value of atoms on the repaired program vocabulary. Let M be the model obtained from M' by reverting transformation in Lemma 16. Then, M is a model of $Why_{\Pi}(L)$ for each $L \in M'$.* ∘

Proof of Lemma 17. Take a model s.t. $M' \in AS(W(\Pi))$. We know by Lemma 15 that M' is the least PSM of repaired program Π' which implies that it is the WFM of program Π'. Now, because $L \in WFM(\Pi')$, we know from Theorem 8.1.1 that there is a conjunction of literals $C \in M$ s.t. $C \models Why_{\Pi}(L)$, and because $C \subseteq M$, it is also true that $M \models Why_{\Pi}(L)$. □

Example 8.2.1 *Recall now Example 8.1.2, to which we apply transformation $W(\Pi)$ where*

$$\pi_{in}(\Pi) = \{ \quad head(r_1,a).\ bodyP(r_1,b).\ bodyP(r_1,c).\ head(r_2,b).\ bodyP(r_2,d).\\ head(r_3,b).\ bodyN(r_3,e).\ head(f_1,c).\ head(f_2,d).\}$$

$W(\Pi)$ has 256 answer sets corresponding to all possible changes to Π by removing or keeping rules, and adding or not facts. Of these answer sets, 6 correspond to literal a being false and all other atoms true, and are in exact correspondence with the propositional models of formula $not(a) \wedge \neg r_1 \wedge (r_2 \vee \neg not(b)) \wedge c \wedge d \wedge \neg not(e)$ obtained in Example 8.1.5. All answer sets below contain the following set of facts[1]*:*

$$\{noFact(a),\ removeRule(r_1),\ keepRule(f_1;f_2),\ missingFact(e)\}$$

which correspond to conjuncts $not(a), \neg r_1, c, d, \neg not(e)$ of the previous formula and are filtered for readability:

$$\{keepRule(r_2;r_3), missingFact(b)\}$$
$$\{keepRule(r_2;r_3), noFact(b)\}$$
$$\{keepRule(r_2), removeRule(r_3), missingFact(b)\}$$
$$\{keepRule(r_2), removeRule(r_3), noFact(b)\}$$
$$\{removeRule(r_2), keepRule(r_3), missingFact(b)\}$$
$$\{removeRule(r_2;r_3), missingFact(b)\}$$

There are 151 possible AS explaining why a is true, corresponding to the 151 provenance models for a: $Why_{\Pi}(a)$. △

8.3 Provenance Transformation for the Answer Set Semantics

Forbidding undefined atoms in the model and also disallowing models where $tu/1$ occurs and $t/1$ does not occur (as in Figure 8.7), adapts the WF transformation presented before to the answer set semantics. We call this meta-transformation $\pi_{as}(\Pi)$ or simply $S(\Pi)$.

$$\pi_{as}(\Pi) = \pi_{in} \cup \pi_{common} \cup \pi_{wfs}(\Pi) \cup \{ \quad \leftarrow\ atom(A),\ tu(A),\ not\ t(A).\\ \leftarrow\ atom(A),\ undef(A).\}$$

Figure 8.7: Meta-transformation $\pi_{as}(\Pi)$ or simply $\mathbf{S}(\Pi)$

We need to define next an auxiliary notion of what is the why not provenance of an intended AS:

[1] We denote a set of facts $\{a(X),...,a(Y)\}$ as $a(X;...;Y)$.

Definition 8.3.1 (Why-not provenance for an interpretation) *Let* Π *be a logic program and I an interpretation. The* **AS WnP** *for I is:*

$$AnsWhy^{I}_{\Pi} = \bigwedge_{\forall a \in I} AnsWhy_{\Pi}(a) \bigwedge_{\forall a \notin I} AnsWhy_{\Pi}(not\ a)$$

▲

Intuitively, the provenance formula for an interpretation I is the intersection of the provenance formulae for its positive (and negative) atoms. We then select provenance formulae containing $\neg r$ for every $r \in \Pi$ such that an atom $A \notin I$ belongs to $Head(r)$ which effectively avoids considering rules giving support to unintended atoms, and thus providing unnecessary justifications.

We define next what we consider to be the concept of Justifications for Answer Set Programming, starting with an auxiliary concept of *exhaustive justifications* which are justifications such that:

(i) They contain atoms in the vocabulary of repaired programs that are true in the answer set of the WFM transformation

(ii) They model the why-not provenance formulae computed for each of their literals.

Definition 8.3.2 (Exhaustive Justifications) *Given a program* Π*, an answer set* $M \in AS(\Pi)$ *and a literal* $L \in M$*, we define an exhaustive justification for literal L as the conjunction C of all atoms in reparation formula* $Rep(M)$ *that are true in* $AS(S(\Pi))$*.* ▲

Definition 8.3.3 (Justifications for Answer Set Programming) *Let* Π *be a logic program and* $AS(S(\Pi))$ *be its provenance answer sets. We define the prime implicants of the union of the exhaustive justifications for these answer sets as being the* justifications *for the answer sets of the original program* Π*.* ▲

We recall that a prime implicant of a function is an implicant that cannot be covered by a more general (more reduced, i.e., with fewer literals) implicant.

The concept in Definition 8.3.3 forms a restricted class, containing interesting WnP formulas. The following Theorems 8.3.1 and 8.3.2 follow from the above Lemmata 15, 16 and 17 for the well-founded semantics:

Theorem 8.3.1 (Provenance Models for the Answer Set Semantics) *Given a logic program* Π*, and an interpretation I, each answer set AS of transformed program* $S(\Pi)$ *can be transformed into another answer set* AS' *corresponding to a model* $M \in AnsWhy^{I}_{\Pi}$ *(Definition 8.3.1), by:*

1. *Replacing every* r_i *of a non-applicable rule with* $\neg r_i$ *for literals not belonging to the interpretation.*

2. *Applying Theorem 8.1.2 to literals L in a conjunction of literals C (i.e.,* $L \in C$*), s.t.* $L = false$ *and L is also undefined in* $WFM(P_{rec}(AS'))$*.*

○

Proof of Theorem 8.3.1.

1. By removing rules that are not applicable (substituting $keepRule(r)$ by $removeRule(r)$ in AS obtaining AS') we obtain an answer set of $S(\Pi)$ by selecting a different choice in π_{Rules} for r; nothing else changes in the model. The obtained repaired program is two-valued and therefore by Lemma 17 there is a model of $M \models Why_{\Pi}(L)$ for each $L \in M$. Since for every literal L in I it is the case that either $Why_{\Pi}(L)$ or $Why_{\Pi}(not\ L)$ then for each literal $AnsWhy_{\Pi}(L)$, and therefore it is also the case that $AnsWhy_{\Pi}(I)$

2. First note that program $S(\Pi)$ is simply imposing extra integrity constraints on $W(\Pi)$, thus any answer set of $S(\Pi)$ is an answer set of $W(\Pi)$. If we add to Π an extra rule $i \leftarrow a_1,\ldots,a_m,\ not\ b_1,\ldots,not\ b_n$ (where i is a new atom, $a_1,\ldots,a_m$ are the true literals in interpretation I and $b_1,\ldots,b_n$ are the false literals in I) and consider only the answer sets containing r_i and $not\ i$ the answer sets of the altered program for which i holds are in exact correspondence with the answer sets of $S(\Pi)$ for which I holds.

   According to Theorem 8.3.2 it is possible to construct the corresponding model of $AnsWhy(i)$ which is immediate to see that is also a model of $AnsWhy(I)$.

Theorem 8.3.1 shows that we may get a justification that does not occur directly as a model of the WnP formula but can be turned into a model in two different ways. Example 8.3.2 illustrates this situation.

Theorem 8.3.2 (From Models of Provenance Formulae to Answer Sets of $S(\Pi)$) *Given a logic program Π and an interpretation I, every model of its provenance propositional formula $AnsWhy^I_{\Pi}$ has a correspondence with an answer set of transformed program $S(\Pi)$.* ○

Proof of Theorem 8.3.2. Let Π' be the program with the additional rule for i, as in the previous theorem. Note that the models of $AnsWhy_{\Pi'}(i)$ will correspond exactly to models of $AnsWhy^I_{\Pi}$ except that propositional variables for rule i may appear. We will consider again those models M' of $AnsWhy_{\Pi'}(i)$ such that $M' \models r_i \wedge not(r_i)$. Now by Lemma 16, we will have an answer set A' of $W(\Pi')$ such that $t(i) \in A'$ and $tu(i) \in A'$. By construction of $W(\Pi')$ it is the case that also $t(a_j) \in A'$ and $tu(a_j) \in A'$ for every $a_j \in I$, and $t(b_k) \notin A'$ and $tu(b_k) \notin A'$ for every $b_k \notin I$. But this means that A' is also an answer set of $S(\Pi')$. Since the answer sets of $S(\Pi')$ for which i, r_i, and $not\ i$ are true in exact correspondence to the answer sets of $S(\Pi)$ for which all literals of I hold, then by removing any atom true in A' which have in arguments i or r_i we obtain an answer set of $S(\Pi)$.

These models are exhaustive in the sense they provide all possible justifications for an AS or all explanations for why an interpretation is not a model. These can then

be minimised according to whatever criteria one might have, e.g., subset minimality, minimal changes to the program, disallowing or preferring certain repair operations over others etc., which can be captured by optimisation constraints supported by the major ASP solvers.

Example 8.3.1 *Consider now the program in Example 1 of Viegas Damásio et al. (2013):*

$$P = \left\{ \begin{array}{l} r_1 : a \leftarrow c, not\ b. \\ r_2 : b \leftarrow not\ a. \\ r_3 : d \leftarrow not\ c, not\ d. \\ r_4 : c \leftarrow not\ e. \\ r_5 : e \leftarrow f. \\ r_6 : f \leftarrow e. \end{array} \right\}$$

This program contains a self-supported loop between atoms e and f. Atom c acts as a guard for that loop, and is true iff the loop produces no results being the case that there is an integrity constraint eliminating models which do not contain the guard c. This atom c is then required, so to say, to activate the choice loop between a and b, hence the program has answer sets $A_1 : \{a,c\}$ and $A_2 : \{b,c\}$. Below are some of the 144 WnP models for A_1 from which we select the ones presenting intuitive explanations (model selection is clarified in the next section) from all of which the following literals are omitted:

$$F = \{keepRule(r_2;\ r_3;\ r_5;\ r_6),\ noFact(b;\ d;\ e;\ f)\}$$

1. $F \cup \{removeRule(r_1),\ keepRule(r_4),\ missingFact(a;c)\}$
2. $F \cup \{removeRule(r_1;\ r_4),\ missingFact(a;c)\}$
3. $F \cup \{removeRule(r_1),\ keepRule(r_4),\ missingFact(a),\ noFact(c)\}$
4. $F \cup \{keepRule(r_1),\ removeRule(r_4),\ noFact(a),\ missingFact(c)\}$
5. $F \cup \{keepRule(r_1;\ r_4),\ missingFact(a;\ c)\}$
6. $F \cup \{keepRule(r_1;\ r_4),\ noFact(a),\ missingFact(c)\}$
7. $F \cup \{keepRule(r_1;\ r_4),\ missingFact(a),\ noFact(c)\}$
8. $F \cup \{keepRule(r_1),\ removeRule(r_4),\ missingFact(a;\ c)\}$
9. $F \cup \{keepRule(r_1;\ r_4),\ noFact(a;\ c)\}$

△

Keeping in mind that our approach captures program repairs, which can be viewed as transformations that bring about a certain answer set, we compare our results next with an example taken from Cabalar & Fandinno (2017), and use it to show the way

prime implicants of these repair answer sets can be regarded as being the most informative way to present these alternatives. We recall that we call *Justifications* to these prime implicants, as per Definition 8.3.3.

Example 8.3.2 (From Cabalar & Fandinno (2017)) *Let P_{Cab} be the program consisting on the following rules:*

$$P_{Cab} = \left\{ \begin{array}{ll} r_1: & a \leftarrow not\ b. \\ r_2: & b \leftarrow not\ a,\ not\ c. \\ f_1: & c. \\ r_3: & c \leftarrow a. \\ r_4: & d \leftarrow b,\ not\ d. \end{array} \right\}$$

In their approach, the (standard) WFM of program P_{Cab} is two-valued and corresponds to the unique (standard) stable model $\{a,c\}$. Furthermore, they obtain two justifications of c with respect to this unique stable model: the fact c and the pair of rules r_1 and r_3. Note that when c is removed $\{a,c\}$ is still the unique stable model, but all atoms are undefined in the $WFM(P_{Cab})$. Hence, $r1$ and $r3$ is a justification for the unique stable model of the program but not with respect to its WFM.

As for our transformation, excluding model Π_{prune}, for P_{Cab} produces the 36 answer sets for interpretation $I = \{a,c\}$. Note that it is the case that these answer sets correspond to the answer set of the original program and thus, as expected, contain no error-indicating atoms but rather present the changes to the program that would still allow us to obtain that answer set. Calculating their prime implicants (with the Quine-McCluskey Method), one obtains the following justifications (presented as a Boolean formula)

$$\begin{array}{rl} & KeepRule(f_1)\ .\ MissingFact(a)\ .\ noFact(b)\ .\ noFact(b) \quad \vee \\ & KeepRule(r_3)\ .\ MissingFact(a)\ .\ noFact(b)\ .\ noFact(b) \quad \vee \\ Imp_1 = & KeepRule(r_1)\ .\ KeepRule(f_1)\ .\ noFact(b)\ .\ noFact(b) \quad \vee \\ & KeepRule(r_1)\ .\ KeepRule(r_3)\ .\ KeepRule(r_4)\ .\ noFact(b)\ .\ noFact(c) \quad \vee \\ & KeepRule(r_1)\ .\ KeepRule(r_2)\ .\ KeepRule(r_3)\ .\ noFact(b)\ .\ noFact(c) \end{array}$$

If we remove the fact for c from the program (rule $f1$, for fact 'c'), we then also obtain 36 answer sets which as expected, contain no error-indicating atoms but rather also present the changes to the program that would still allow us to obtain that answer set. Calculating their prime implicants, one obtains the following justifications (presented as a Boolean formula):

$$\begin{array}{rl} & MissingFact(a)\ .\ MissingFact(c)\ .\ noFact(b)\ .\ noFact(b) \quad \vee \\ Imp_2 = & KeepRule(r_3)\ .\ MissingFact(a)\ .\ noFact(b)\ .\ noFact(b) \quad \vee \\ & KeepRule(r_1)\ .\ MissingFact(c)\ .\ noFact(b)\ .\ noFact(b) \quad \vee \\ & KeepRule(r_1)\ .\ KeepRule(r_3)\ .\ noFact(b)\ .\ noFact(b) \end{array}$$

The provenance formula for the corresponding answer set is:

$$[r1 \wedge not\ b \wedge (\neg r2 \vee \neg not\ a \vee c) \vee \neg not\ a] \wedge$$
$$\neg\,[r2 \wedge not\ a \wedge \neg c \vee \neg not\ b] \wedge$$
$$[(r3 \wedge r1 \wedge not\ b \wedge \neg r2) \vee (r3 \wedge \neg not\ a) \vee c] \wedge$$
$$\neg[r4 \wedge not d \wedge (r2 \wedge not\ a \wedge \neg c \vee \neg not\ b) \vee \neg not\ d]$$
$$\equiv$$
$$[r1 \wedge not\ b \wedge (\neg r2 \vee \neg not\ a \vee c) \vee \neg not\ a] \wedge not\ b \wedge [\neg r2 \vee \neg not\ a \vee c] \wedge$$
$$[(r3 \wedge r1 \wedge \neg r2) \vee (r3 \wedge \neg not\ a) \vee c] \wedge not d \wedge [\neg r4 \vee \neg not\ d \vee not\ b \wedge (\neg r2 \vee \neg not\ a \vee c)]$$

This formula can be further simplified but we can already see that a model of the prime implicants formula having:

$$r1 \wedge r2 \wedge r3 \wedge r4 \wedge not\ a \wedge not\ d \wedge \neg c \wedge not\ b$$

is not a model of the provenance formula, but we can also see that

$$r1 \wedge \neg r2 \wedge r3 \wedge r4 \wedge not\ a \wedge not\ d \wedge \neg c \wedge not\ b$$

is indeed a model of the formula (assuming b as false because the corresponding rule is not applicable).

△

8.4 Unifying Provenance with Debugging

As shown before, our meta-transformation produces a model for each WnP model and some can be aligned with debugging models calculated by Gebser *et al.* (2008). These approaches complement each other: we produce provenance models for existing answer sets, while the debugging approach is capable of obtaining more specific results regarding the non-existence of answer sets, namely in the presence of unfounded loops. So, we need to impose equivalence between predicates $int/1$ and $t/1$ (see Fig. 8.8) and thus we introduce a new module π_{map}. The resulting models consist of two parts, one stating what is the problem with the interpretation at hand (corresponding to the debugging part) and the other offering a justification for why this interpretation is a model of the program (corresponding to the provenance part).

Module π_{t-int} ensures that atoms $int/1$ and $t/1$ are equivalent which effectively maps *provenance* and *debugging* at the interpretation level, while π_{ics} guarantees that violated ICs are corrected by removing them. The combined program is:

$$J(\Pi) = \pi_{int} \cup \pi_{sat} \cup \pi_{supp} \cup \pi_{ufLoop} \cup \pi_{as}(\Pi) \cup \pi_{map} \cup D'(\Pi)$$

$$\pi_{t-int} = \left\{ \begin{array}{l} \leftarrow atom(A),\ int(A),\ not\ t(A). \\ \leftarrow atom(A),\ \overline{int}(A),\ t(A). \end{array} \right\}$$

$$\pi_{ics} = \left\{ \leftarrow rule(R),\ violated(R),\ keepRule(R). \right\}$$

$$\pi_{prune} = \left\{ \begin{array}{ll} \leftarrow & rule(R),\ removeRule(R),\ not\ ap(R). \\ \leftarrow & atom(A),\ missingFact(A), \\ & supported(A),\ not\ ufLoop(A). \\ \leftarrow & atom(A),\ noFact(A),\ supported(A),\ ufLoop(A). \end{array} \right\}$$

Figure 8.8: Transformation $\pi_{map} = \pi_{t-int} \cup \pi_{ics} \cup \pi_{prune}$

where in order to determine provenance for a given AS, $D'(\Pi)$ is obtained from $D(\Pi)$ by substituting π_{ap} with π_{apttu} and removing π_{noas}.

Intuitively, an interpretation is guessed (represented by $int/1$), and one then forces the correspondence of $t/1$ with $int/1$. The **repaired program** (removing rules or adding missing facts) is guessed, and generates the extension of $t/1$, and it is always possible to trivially repair a program and obtain any desired interpretation by removing all rules and adding all missing facts as prescribed by the corresponding repair predicates. We now look at error-indicating predicates to detect problems with Π.

Theorem 8.4.1 *Let M be an AS of $D(\Pi)$. Then, there is an AS M' of meta-program $J(\Pi)$ such that*

$$M \setminus (\{noAnswerSet\} \cup \{ap(r_i), bl(r_i) \mid r_i \text{ is a rule identifier}\}) \subseteq M'$$

∘

Proof of Theorem 8.4.1. We retain the two procedural modules of Gebser *et al.* (2008), namely the ones detecting loops and violation of integrity constraints, while mapping their other modules that detect unsupported literals ($unsupported(X)$) and unsatisfied rules ($unsatisfied(R)$) respectively with provenance signalling that a fact is missing ($missingFact(X)$) and that a rule must be removed ($removeRule(R)$). We furthermore impose in π_{map} that the basic interpretation literals int and t must be mutually consistent which effectively substantiates our meta transformation as an extension to their setting and thus indexes our models to their models at the level of truth assignments for literals in the interpretation. Because of this it suffices to observe that, if one looks only at the non-disjunctive part of the corresponding rules, the conditions in which $unsupported(X)$ occurs are the same as the ones where $missingFact(X)$ occurs and the same goes for $unsatisfied(R)$ with respect to $removeRule(R)$. □

So, we are able to detect every error pointed out by error-indicating predicates of Gebser *et al.* (2008). There is however a subtle difference: we prune debugging answer sets which are not supported by the repaired program. Their exact relationship is captured next.

Theorem 8.4.2 (Mapping) *Let M' be an answer set of $J(\Pi)$. Then,*

- *If $unsatisfied(r_i)$ or $violated(r_i) \in M'$ then $removeRule(r_i) \in M'$;*
- *If $unsupported(r_i) \in M'$ then $missingFact(r_i) \in M'$;*
- *If $ufLoop(a_1..a_n) \in M'$ then $\exists i \in [1,\ldots,n]$ such that $missingFact(a_i) \in M'$. Also,*
  $\exists M'' \in AS(J(\Pi))$ s.t. $ufLoop(a_1..a_n) \cup missingFact(a_1..a_n) \in M''$.

$\circ$

Proof of Theorem 8.4.2. This is necessarily the case because the $t/1$ and $int/1$ literals are mapped one to one by module π_{map} which forces the literals to correspond exactly as stated in the theorem. $\square$

However, some provenance answer sets may be considered redundant (even though correct) and we present module π_{prune} in Fig. 8.8, which can be used to prune these. It disallows removing blocked rules ($bl/1$), adding facts which are not in unfounded loops but are already supported, and forces a *missingFact* to be added for at least one atom belonging to each detected unfounded loop.

Example 8.4.1 *Take again the program in Example 8.3.1 and include relevant modules of transformation D. We show next a sample of its answer sets, having in common the following set of facts:*

$$F = \left\{ \begin{array}{l} keepRule(r1;r2;r3;r4;r6), \\ unsupported(a;b), \\ missingFact(a;b), \\ noFact(c;e) \end{array} \right\}$$

$$F \cup \{\ removeRule(r_5), unsupported(d;f), unsatisfied(r_5), missingFact(d;f)\ \}$$

$$F \cup \left\{ \begin{array}{c} removeRule(r_5), unsatisfied(r_5), supported(c;e), noFact(d), \\ unsupported(f), missingFact(f) \end{array} \right\}$$

$$F \cup \{\ keepRule(r_5), supported(c), unsupported(d), missingFact(d), noFact(f)\ \}$$

$$F \cup \{\ keepRule(r_5), supported(c), noFact(d;f)\ \}$$

$\triangle$

8.5 Conclusions and Future Work

We provide a transformation to compute provenance models under the well founded and answer set semantics by computing the answer sets of meta-programs that capture the original programs and include some necessary extra atoms. We do this while preserving compatibility with the previous work of Viegas Damásio *et al.* (2013) and computing the models directly without first obtaining the provenance formulas for certain interpretations. This enables computing provenance answer sets in an easy way by

using answer set solvers. We then introduce the concept of justifications as being the prime implicants of these answer sets. Having this, we align provenance and debugging answer sets in a unified transformation and show that the provenance approach generalises the debugging one, since any error has a counterpart provenance but not the other way around. Since the proposed method is based on meta-programming, we extended an existing tool Gebser *et al.* (2007b) and developed a proof-of-concept (http://cptkirk.sourceforge.net) built solely to allow computing our models.

Our mapping allows generating answer sets capturing errors and justifications for (intended) models. As expected, they are exponential. One direction to explore is to obtain prime implicants by optimising these models using reification and then subset inclusion preference ordering Gebser *et al.* (2007a, 2011b) via a saturation technique Eiter & Gottlob (1995b). Note that deciding if an AS is optimal for some disjunctive logic program is a Π_2^p-complete problem. Alternative *offline justifications* Pontelli *et al.* (2009) (which are also exponential) can be extracted from models of $J(\Pi)$ by adding extra constraints to the transformed program guaranteeing: only one rule is kept for true atoms (providing support); literals assumed false have all rules removed (which are undefined in the WFM); false literals have to keep all their rules; and the dependency graph is acyclic. The major difference to Pontelli's approach is that we provide justifications for the full model from which we may obtain their justifications, but our approach subsumes it since we are capable of finding more justifications as well as errors in the program.

Acknowledgements We would like to thank Javier Romero and Torsten Schaub for the helpful suggestions on how to perform subset minimisation over disjunctive logic programs using the *metasp* system. We would also like to thank Pedro Cabalar and Jorge Fandiño for the exchange of ideas on the complexity of finding why-not formulas for literals. We also thank Martin Gebser for providing a simple way to calculate the prime implicants (as a propositional formula) given a set of answer sets.

Chapter 9

Application Scenario: Characterising Conflicts in Access Control Policies

The emergence of technologies such as service-oriented architectures and cloud computing, nowadays, allows us to perform business services more efficiently and effectively. Access control is an important mechanism for achieving security requirements in such information systems, which are described by means of access control policies (ACPs).

However, these security requirements cannot be guaranteed when conflicts occur in these ACPs. Furthermore, the design and management of access control policies is often error-prone due not only to the lack of a logical and formal foundation but also to the lack of automated conflict detection and resolution.

We use a meta-model $\mathcal{M}^P$ Barker (2010) to describe ACPs as logic programs, identify their basic conflict types and characterise them in terms of (relativised) strong equivalence of logic programs. This characterisation allows for the automatic identification of such conflicts among other reasoning tasks.

9.1 Introduction

We begin by clarifying the term *policy* in the context of access control which is somewhat ambiguous in available literature. Next we describe different aspects of access control policies that are of general interest to us and serve as motivation for our approach.

Still in this section we describe the state of the art relevant for us in the area of logic based access control, particularly meta-models for access control, keeping in mind the way it reflects in answer set programming Gelfond & Lifschitz (1988) in particular.

We also present an overview of the relevant work in the context of logic programs

as well as, towards the logical characterisation of conflicts in access control, an introduction to strong equivalence in the context of logic programs due to Lifschitz *et al.* (2000) and a relativised version of this notion due to Eiter *et al.* (2007).

In the following section we start by identifying and characterising different access control conflicts. We then discuss the interplay of exceptions in access control, particularly with our translation to answer set programming, with default negation in logic programming. After this, there is a section where we discuss conflict resolution methods and end with conclusions and future work.

9.1.1 Hierarchies, Inheritance and Exceptions

For a long time now, computer security models have supported some forms of abstraction regarding the authorisation elements, to formulate security policies concisely. For example, users can be organised in groups. The authorisations granted to a user group is applicable to all of its member users, and authorisations concerning a class of objects apply to all of its member objects. This is typically modelled via an **authorisation hierarchy** derived from the hierarchies of subjects, resources, and operations (basic hierarchies).

The authorisation hierarchy can be exploited to formulate policies in a incremental and top-down fashion. Starting with an initial set of general authorisations that can be progressively refined with more specific authorisations that in turn introduce exceptions to the general rules. A benefit that come together with this is that policies may be expressed concisely and allow easy management. Exceptions make inheritance a defeasible inference in the sense that **inherited authorisations** can be retracted (or overridden) as exceptions are introduced. As a consequence, the underlying logic must be non-monotonic.

Exceptions require richer authorisations. It must be possible to say explicitly whether a given permission is granted or denied. Then authorisations are typically extended with some form of sign for granted permissions and some form of negation for denials. It may easily happen that two conflicting authorisations are inherited from two incomparable authorisations, therefore a policy specification language featuring inheritance and exceptions must necessarily deal with conflicts.

A popular conflict resolution method — called denial takes precedence — consists of overriding the positive authorisation with the negative one (i.e., in case of conflicts, authorisation is denied), but this is not the only possible approach. In Al-Kahtani & Sandhu (2004) the analysis includes user authorisation, conflict detection among rules, conflict resolution polices, the impact of negative authorisation on role hierarchies and an enforcement architecture.

Recent proposals have worked towards languages and models that are able to express, in a single framework, different inheritance mechanisms and conflict resolution policies. Logic-based approaches, so far, are the most flexible and expressive.

9.1.2 Access Control Policies

For a long time now, logic programming and rule-based reasoning have been proposed as a strong basis for policy specification languages. However, the term "policy" has never been given a unique meaning. In fact, it is used in the literature in an ambiguous and broad sense that encompasses at least the following types:

Access Control Policies are policies that pose constraints on the behaviour of a system. They are typically used to control permissions of users/groups while accessing resources and services.

Trust Management is based on policy languages used to collect user properties in open environments, where the set of potential users spans over the entire web and by definition is a priori partially unknown.

Action Languages are used in the specification of reactive policies to execute actions like event logging, notifications, etc. Authorisations that involve actions and side effects are sometimes called provisional.

Action languages typically are sorted into two classes: action description languages and action query languages. Examples of the former include STRIPS, PDDL, Language A (a generalisation of STRIPS), Language B (an extension of A) and Language C (which adds indirect effects also, and does not assume that every fluent is automatically "inertial").

There are also the Action Query Languages P, Q and R, being the case that there are conversions from these languages into ASP, from which we particularly highlight the translation for action language C.

Business Rules are "statements about how a business is done". These are used to formalise and automate business decisions as well as for efficiency reasons. They can be formulated as reaction rules, derivation rules, and integrity constraints.

We will use the term Policy as being an Access Control Policy.

In Kolovski (2007); Bonatti *et al.* (2009), the reader can find good introductory surveys to logic-based ACPs. Kolovski (2007) is limited to the presentation of a DL-based formalism to represent XACML[1] policies, which is not formally characterised, while Bonatti *et al.* (2009) considers a more general overview, including XACML.

Motivation The need for characterising conflict factors is not only due to the non-monotonic nature of access control in ASP but also to the fact that some policies are distributed, for instance in the form of different logic programming modules, and

[1] XACML stands for "eXtensible Access Control Markup Language". The standard defines a declarative fine-grained, attribute-based access control policy language, an architecture, and a processing model describing how to evaluate access requests according to the rules defined in policies.

their interplay can as such derive conflicting conclusions. The characterisations we present have the potential to improve the usage of answer set programming in an access control framework by adding a further level of policy assurance.

We present next an example of an access control policy where a user is represented by its credentials, each serving different purposes and each with different attributes. The example contains only one user and one credential to improve readability. Each of the aforementioned attributes need to be trusted by an entity in order to be acceptable for a certain purpose. Subscriptions are available for different resources. There are then also rules for deciding whether a user is authenticated, whether a credential is valid, or to check if the policy grants access to a resource for a given purpose.

Example 9.1.1 presents a very simple access control policy in the form of a positive logic program where the the choice rule could, in the context of this positive program, be replaced by $selectCred(X) \vee nselectCred(X) \leftarrow credential(X)$.

Several limitations arise from directly implementing ACPs as positive logic programs. One comes from the fact that it is important to have a meta-model to describe and possibly (inter)change this policy. Others result from the fact that more expressive power is needed and features such as default knowledge and exceptions are desirable and can be included, thus making it non-monotonic.

Negative authorisation is also described in the literature as being a necessary feature which also leads to problems.

Example 9.1.1 *The following is an example of an access control policy in ASP* [2]*:*

$$
\begin{aligned}
allow(download, Resource) :- \quad & public(Resource).\\
allow(download, Resource) :- \quad & authenticated(User),\\
& hasSubscription(User, Subscription),\\
& availableFor(Resource, Subscription).\\
\\
authenticated(User) :- \quad & valid(Credential),\\
& attr(Credential, name, User).\\
\\
valid(Credential) :- \quad & selectCred(Credential),\\
& attr(Credential, type, T),\\
& attr(Credential, issuer, CA),\\
& trustedFor(CA, T).
\end{aligned}
$$

[2]Example 9.1.1 was taken and then adapted from `http://asptut.gibbi.com` Eiter *et al.* (June 2006).

$hasSubscription(\text{“}Joao\text{”}, law\_basic).$
$hasSubscription(\text{“}Joao\text{”}, computer\_basic).$

$availableFor(\text{“}nmr12.pdf\text{”}, computer\_basic).$

$trustedFor(\text{“}NewUniversity\text{”}, id).$
$trustedFor(\text{“}PTGovernment\text{”}, ssn).$

$\%resources\ r(at_1, ..., at_n);$
$credential(cr01).$
$attr(cr01, type, id).$
$attr(cr01, name, \text{”}Joao\text{”}).$
$attr(cr01, issuer, \text{”}NewUniversity\text{”}).$

$\%Choice\ rule\ used\ to\ decide\ if\ a\ credential\ is\ used\ or\ not$
$\{selectCred(X)\} :- credential(X).$

△

This program has two answer sets, from which we filter predicates *selectCred*, *valid*, *authenticated* and *allow*, namely:

$$\begin{aligned} AS_1 = \{ \quad & selectCred(cr01), valid(cr01), authenticated(\text{“}Joao\text{”}), \\ & allow(download, \text{”}nmr12.pdf\text{”})\} \\ AS_2 = \{ \quad & \} \end{aligned}$$

9.1.3 The $\mathcal{M}^P$ model

Over the years, research in access control has proposed a number of different models and languages in which terms authorisation policies can be defined. Despite the variety of proposed access control models described in the literature, most of the existing access control models are based on a small number of primitive notions.

In Barker (2010) the authors describe the interpretation, syntax and semantics that are adopted in their proposed **access control meta-model** $\mathcal{M}^P$, which attempts to identify the aforementioned small number of primitive notions and that we will use throughout this chapter.

In order to define $\mathcal{M}^P$, a prior version called meta-model $\mathcal{M}$ Barker (2009) is extended to accommodate data subjects, data controllers, denials of access, the notion of purpose, contextual accessibility criteria and the flexible specification of permitted recipients of a data subject's personal data. For that, the following core (interpreted) relations of the $\mathcal{M}^P$ model (defined with respect to their many-sorted language) are used:

- PCA, a 4-ary relation.
- ARCA, a 5-ary relation.

- ARCD, a 5-ary relation.
- PAR, a 3-ary relation.
- PRM, a 3-ary relation.

The semantics of the n-ary tuples in *PCA*, *ARCA*, *ARCD*, *PAR*, and *PRM* are, respectively, defined as:

- $(k_{ds}, k_{du}, c, p) \in PCA$ if and only if a data user $k_{du} \in K_{du}$ is assigned to the category $c \in C$ for the purpose $p \in P$ according to the data subject $k_{ds} \in K_{ds}$.
- $(k_{ds}, a, r, c, p) \in ARCA$ if and only if the permission (a, r), with respect to action $a \in A$ and resource $r \in R$, is assigned to the category $c \in C$ for the purpose $p \in P$ according to the data subject $k_{ds} \in K_{ds}$.
- $(k_{ds}, a, r, c, p) \in ARCD$ if and only if the permission (a, r), with respect to action $a \in A$ and resource $r \in R$, is denied to the category $c \in C$ for the purpose $p \in P$ according to the data subject $k_{ds} \in K_{ds}$.
- $(k_{du}, a, r) \in PAR$ if and only if a data user $k_{du} \in K_{du}$ is authorised to perform the action $a \in A$ on the resource $r \in R$.
- $(k_{ds}, r, m) \in PRM$ if and only if the data subject $k_{ds} \in K_{ds}$ "controls" access to the resource $r \in R$ and k_{ds} asserts that the meta-policy $m \in M$ applies to access on the resource r.

For representing hierarchies of categories, the following definition is included as part of the axiomatisation of $\mathcal{M}^P$ (where '_' denotes an anonymous variable):

$$
\begin{aligned}
&contains(C, C) \leftarrow dc(C, \_).\\
&contains(C, C) \leftarrow dc(\_, C).\\
&contains(C', C'') \leftarrow dc(C', C'').\\
&contains(C', C'') \leftarrow dc(C', C'''), contains(C''', C'').
\end{aligned}
$$

Authorisation may then be generically defined in $\mathcal{M}^P$ terms as:

$$
\begin{aligned}
par(K_{du}, A, R) \leftarrow \quad & prm(K_{ds}, R, C),\ pca(K_{ds}, K_{du}, C', P),\\
& contains(C, C'), arca(K_{ds}, A, R, C, P).
\end{aligned}
$$

Intuitively meaning that the entity who is responsible for the resource asserts that the policy C applies to access on the resource R, while the data user is assigned to the category C' for the purpose P, category which is contained and C and for which category, permission is assigned for a specific purpose.

In this instance, a closed policy is specified as being enforced by all data subjects, and contains is a definition of a partial ordering of categories that are elements in the transitive-reflexive closure of a "directly contains" (dc) relation on pairs of category

identifiers $dc(c_i, c_j)$, such that: $\Pi \models dc(c_i, c_j)$ if and only if the category $c_i \in C$ ($c_i \neq c_j$) is senior to the category $c_j \in C$ in a category hierarchy defined in Π and there is no category $c_k \in C$ such that $\Pi \models dc(c_i, c_k) \wedge dc(c_k, c_j)$ holds where $c_k \neq c_i$ and $c_k \neq c_j$.

Although the partial ordering of categories is often a feature of access control models, it should be clear that other relationships between categories may be easily defined within the $\mathcal{M}^P$ model.

We point out that this meta-model is formally well defined and is essentially based on the use of just five key interpreted relations (the *pra*, *pca*, *arca*, *arcd* and *prm* relations) and two proper axioms that define *par* and *contains*.

For further details we refer the reader again to Barker (2009, 2010).

Example 9.1.2 (Translation to $\mathcal{M}^P$) *The following is the translation of Example 9.1.1 into $\mathcal{M}^P$ meta-model, encoded as a logic program:*

```
arca(public,download,Resource,all,any):-
    prm(public,Resource,MetaPolicy ).

arca(public,download,Resource,all,SubscriptionType):-
    pca(ds,User,authenticated,generic),
    pca(ds,User,hasSubscription,SubscriptionType),
    par(SubsriptionType,availableFor,Resource).

pca(DS,User,authenticated,generic) :-
     pca(DS,DU,Credential,valid),
     prm(ds, attr(cr01,name,User), metapolicy1).

pca(DS,DU,Credential,valid) :-
    par(DU,selectCred,Credential),
    prm(DS, attr(CR,type,Type),MP),
    prm(DS, attr(CR,issuer, Issuer),MP),
    par(Type,trustedFor,Organisation).

par("Joao",hasSubscription,law_basic).
par("Joao",hasSubscription,computer_basic).

par(computer_basic,availableFor,"nmr12.pdf").

par(id,trustedFor,"New University").
par(ssn,trustedFor,"PT Government").

par(du,credential,cr01).
```

```
prm(ds,attr(cr01,type,id),metapolicy1).
prm(ds,attr(cr01,name,"Joao"),metapolicy1).
prm(ds,attr(cr01,issuer,"New University"),metapolicy1).

%Credential selection
{par(DU,selectCred,Credential)} :- par(DU,credential,Credential).
```

△

9.2 Strong Equivalence of Logic Programs

Towards the logical characterisation of conflicts in access control, we point to the introduction to the logic of here-and-there in Chapter 2 and introduce next the notions of strong and relativised equivalence in the context of logic programs due respectively to Lifschitz *et al.* (2000) and Eiter *et al.* (2007).

Strong Equivalence Theorem A program π is unary if, in every rule of π, the head is an atom and the body is either $\top$ or an atom. In the statement of the theorem, formulas and rules are identified in the sense of nested logic programs Lifschitz *et al.* (1999) without strong negation and with propositional formulas. Accordingly, programs become a special case of theories, and we can talk about the equivalence of programs in the logic of here-and-there.

Theorem 9.2.1 (Lifschitz *et al.* (2000)) *For any programs π_1 and π_2 that are strongly equivalent ($\pi_1 \equiv_s \pi_2$), the following conditions are equivalent:*

(a) *for every program π, programs $\pi_1 \sqcup \pi$ and $\pi_2 \sqcup \pi$ have the same answer sets,*

(b) *for every unary program π, programs $\pi_1 \sqcup \pi$ and $\pi_2 \sqcup \pi$ have the same answer sets,*

(c) *π_1 is equivalent to π_2 in the logic of here-and-there.*

○

The fact that (b) implies (a) shows that the strong equivalence condition we are interested in ("for every π, $\pi_1 \sqcup \pi$ is equivalent to $\pi_2 \sqcup \pi$") does not depend very much on what kind of program π is assumed to be: it does not matter whether π is required to belong to the narrow class of unary programs or is allowed to be an arbitrary program with nested expressions. The fact that (a) is equivalent to (c) expresses the correspondence between the strong equivalence of logic programs and the equivalence of formulas in the logic of here-and-there.

9.2.1 Relativised Notions of Strong and Uniform Equivalence

In what follows, we revise the notion of relativised strong equivalence (RSE) and relativised uniform equivalence (RUE) due to Eiter *et al.* (2007).

Definition 9.2.1 (Relativised Strong and Uniform Equivalence) *Let P and Q be programs and let A be a set of atoms. Then,*

(i) *P and Q are strongly equivalent relative to A, denoted $P \equiv_s^A Q$, iff $P \cup R \equiv Q \cup R$, for all programs R over A;*

(ii) *P and Q are uniformly equivalent relative to A, denoted $P \equiv_u^A Q$, if and only if $P \cup F \equiv Q \cup F$, for all sets of (non-disjunctive) facts $F \subseteq A$.*

▲

Observe that the range of applicability of these notions covers ordinary equivalence (by setting $A = \emptyset$) of two programs P , Q, and general strong (respectively, uniform) equivalence (whenever $Atm(P \cup Q) \subseteq A$). Also the following relation holds:

$$\text{For any set } A \text{ of atoms, } A' = A \cap Atm(P \cup Q). \textit{ Then,}$$

$$P \equiv_e^A Q \text{ holds, if and only if } P \equiv^{A'} Q \text{ holds, for } e \in \{s, u\}.$$

They show that RSE shares an important property with general strong equivalence: In particular, they state that it appears that for strong equivalence, only the addition of unary rules is crucial. That is, by constraining the rules in the set in Definition 9.2.1 to unary rules does not lead to a different concept.

9.3 Conflict types in Access Control and their Characterisation

Prohibition is essential to achieve the security requirements of modern information systems. However, defining prohibition in access control model will give rise to conflicts. A policy conflict occurs when the objectives of two or more policies cannot be simultaneously met. Wang *et al.* (2010) have summarised three types of policy conflicts in their model, defined as modality, redundancy and potential conflicts.

9.3.1 Modality Conflict

Modality conflicts are inconsistencies in the policy specification which may arise when two or more policies with opposite modalities refer to the same authorisation subjects, actions and objects.

Simply put, a modality conflict occurs when there are both allow and deny decisions with the same authorisation subject, action and objects.

Definition 9.3.1 *Let π be an access control policy. We say that there is a modality conflict if:*

$$\pi \models_b Allow(X) \wedge Deny(X)$$

Where $\models_b$ denotes brave consequence. ▲

Due to the introduction of an *arcd* predicate to represent $\neg$ *arca*, in ASP and $\mathcal{M}^P$ terms, a modality conflict means having both *arca* (allow) and *arcd* (deny) predicates with the same arguments in the same model.

9.3.2 Redundancy Conflict

It is known that assigning priorities (which implies an order, be that total or only partial) to access control policies can solve modality conflicts. However, this method can lead to the emergence of policy modules that never apply, which can be called redundant policies. Even though a redundancy conflict has no influence in the enforcement of the access control policies, it should be identified and dealt with because a redundant policy often reflects a mistake that was made while describing security requirements.

Definition 9.3.2 (Redundancy in $\mathcal{M}^P$) *Let π be an access control policy and Π a set of access control policies. Assuming that the priority of Π is higher than the priority of π (i.e., $\Pi \prec \pi$), we have a redundancy conflict in $\mathcal{M}^P$ terms when, for each subject, action and object in π, arca literals are derived such that:*

$$\pi \models_c arca(d_s, a, r, c, p) \text{ and } \Pi \models_c arca(d_s, a, r, c, p)$$

or arcd literals are derived such that:

$$\pi \models_c arcd(d_s, a, r, c, p) \text{ and } \Pi \models_c arcd(d_s, a, r, c, p)$$

▲

Thus, an access control policy π is a redundant policy if a permission or a prohibition, having the same authorisation subject, authorisation action, and authorisation object as π, is always ($\models_c$ denotes cautious consequence)[3] derived from the set of access control policies with higher priority than the priority of π[4].

Definition 9.3.3 (Redundancy in terms of RSE) *Let Π be a set of access control policies and π be a policy such that $\Pi \prec \pi$. A redundancy conflict occurs when:*

$$\pi \cup \Pi \equiv_s^A \Pi$$

Thus, if $\pi \cup \Pi$ is strongly equivalent (relative to A) to Π, where A is the language of $\mathcal{M}^P$.

▲

[3] We consider here the well-known concepts of brave and cautious (skeptical) consequence.

[4] Note that the meta-model $\mathcal{M}^P$ does not allow directly the specification of order between policies but this can be easily done in ASP.

This characterisation of redundancy in terms of strong equivalence has been noted in the past e.g., in Eiter *et al.* (2004).

9.3.3 Potential Conflict

Notice that the above two types of conflict are inconsistencies related to the authorisation subject, action and object. There is another type of conflict between two policies having overlaps in their condition expression. It is the case that there are no modality nor redundancy conflicts between the two polices, but when their associated conditions are simultaneously satisfied, e.g., by adding logic programming modules that satisfy the bodies of specific rules, the two policies result in a modality conflict or redundant conflict. Consequently, potential conflicts are highly pervasive in access control systems.

According to the definition, when some policies have the same condition literals, where a condition is a conjunctive formula $P_1 \wedge \cdots \wedge P_n$ and each P_i represents a generic well-formed formula. One can automatically infer the existence of potential conflicts among these policies.

Definition 9.3.4 (Potential Conflict) *A potential conflict occurs between two policies π_i and π_j in $\mathcal{M}^P$ terms if:*

1. *π_i derives a permission (in the form of an arca literal) and π_j derives a prohibition (in the form of an arcd literal),*

2. *There are overlaps such as: $condition(\pi_j) \cap condition(\pi_i) \neq \emptyset$, and*

3. *There is no policy π_k in the policy set such that*

$$condition(\pi_i) \wedge condition(\pi_j) \rightarrow condition(\pi_k)$$

and π_k derives a prohibition when

$$priority(\pi_i) \prec priority(\pi_j) \prec priority(\pi_k)$$

or π_k derives a permission when

$$priority(\pi_j) \prec priority(\pi_i) \prec priority(\pi_k).$$

▲

Potential conflicts have also been analysed in the extension of Lobo's PDL with ordered disjunction by Bertino (2005), where logic programming with ordered disjunction was used as a way to prioritise action execution in case conflicting actions were triggered by the policy.

9.4 Default Negation as a Cause of Conflicts

Throughout this section we will present examples of programs which we start by formulating as sets of default rules and then present their translation into ASP and their underlying characterisation. Knowledge is represented in *default logic* Reiter (1987) by a default theory $\langle D, W \rangle$ consisting of a set of defaults D and a set of formulas W. Each default rule like:

$$\frac{A : B_1, \ldots, B_n}{C}$$

where A is a prerequisite, $B_1 \ldots B_n$ are justifications and C is a conclusion, is represented in LP as a rule

$$C \leftarrow A,\ not\ \neg B_1, \ldots,\ not\ \neg B_n.$$

There is work in the literature about intuitionistic interpretations of default logic Cabalar & Lorenzo (2004). In Woo & Lam (1992), default rules are used to provide semantics to closed and open policy bases for the case where a policy is represented as a 4-tuple $A = (P^+, P^-, N^*, N^-)$ (akin to Routley models) into which explicit approvals and denials as well as undetermined decisions can be fitted. We leave a model theoretic characterisation of conflicts in terms of these Routley models (or SE-models) for future work.

9.4.1 Characterising Conflicts in Terms of Default Theories

Some conflicts in a default theory cause the non-existence of answer sets. We call such conflicts inconsistencies, incoherencies and conflicts. A default theory is conflicting if it has no default extension.

Conflicts may be categorised into four different types, namely: *default rule with exception*, *omission of mandatory choice*, *need for action rules* and *incoherence: odd negative loops*, which we define and then capture in the examples, are presented next. We also show the way in which they can be reflected as access control conflicts translated to conflicting ASP programs and the way to capture them with the aforementioned $\mathcal{M}^P$ meta-model:

Default Rule with Exception A common conflict in the context non-monotonic knowledge bases occurs when there is a default rule and an exception, in the form of a fact or some other rule, that contradicts it.

Definition 9.4.1 (Default rule with exception.) *Let $T = \langle D, W \rangle$ with $D = \{\frac{:B}{\neg A}\}$ and $W = \{A\}$ be a default theory. In T, a contradiction occurs between W and the consequents of applicable defaults.* ▲

Example 9.4.1 *Consider that Pedro is applying for a new role as software security validation engineer. The company for which he is applying has a rule forbidding access to its code to non employees. However, as part of the interviewing process,*

the company wants to allow Pedro access the code. The following captures this as an ASP program reflecting this conflict:

$$P_1 = \left\{ \begin{array}{l} allow\_analysis(pedro, code). \\ candidate(pedro). \\ \neg allow\_analysis(X, code) \leftarrow \ not\ employee(X). \end{array} \right\}$$

△

Where the following is its translation to $\mathcal{M}^P$:

$$P_1^{\mathcal{M}^P} = \left\{ \begin{array}{l} arca(ds, read, code, pedro, analysis). \\ pca(ds, pedro, candidate, analysis). \\ arcd(ds, read, code, pedro, analysis) \leftarrow \\ \qquad not\ pca(ds, pedro, employee, analysis). \end{array} \right\}$$

Considering the $\mathcal{M}^P$ meta-model, Example 9.4.1 derives a contradiction because $\mathcal{M}^P \models_b arca(ds, read, code, pedro, analysis) \wedge arcd(ds, read, code, pedro, analysis)$. and hence, its only answer set contains these two contradicting atoms:

$$arca(ds, read, code, pedro, analysis)$$

and

$$arcd(ds, read, code, pedro, analysis)$$

Omission of Mandatory Choice When two different default rules contradict each other, an exception (or a choice), must be made between one of the the two.

Definition 9.4.2 (Omission of Mandatory Choice) *Let $T = \langle D, W \rangle$, where $D = \{\frac{:B}{C}, \frac{:D}{\neg C}\}$ and $W = \emptyset$ be a default theory. In T, conflicts occur in the consequents of applicable defaults.* ▲

Example 9.4.2 *In its knowledge base, the same company refers to persons as employees or candidates.*

A company might want to allow access to its premises if it does not explicitly know that someone is not an employee and conversely, allow access to its premises if it does not explicitly know that someone is a candidate. Pedro is a person, known to the company, but for some reason he is not registered in the knowledge base either as an employee or as a candidate.

The following is an example of an ASP program reflecting this conflict:

$$P_2 = \left\{ \begin{array}{l} person(Pedro). \\ allow\_entrance(X, premises) \leftarrow \ person(X),\ not\ \neg employee(X). \\ \neg allow\_entrance(X, premises) \leftarrow \ person(X),\ not\ \neg candidate(X). \end{array} \right\}$$

△

Where the following is its (simplified) translation to $\mathcal{M}^P$:

$$P_2^{\mathcal{M}^P} = \left\{ \begin{array}{l} arca(ds, enter, premises, X, general) \leftarrow \\ \qquad\qquad not\ \neg pca(ds, X, employee, general). \\ arcd(ds, enter, premises, X, general) \leftarrow \\ \qquad\qquad not\ \neg pca(ds, X, candidate, general). \end{array} \right\}$$

Considering the $\mathcal{M}^P$ meta-model, Example 9.4.2 presents a contradiction because its only answer set contains *arca* and *arcd* predicates with the same arguments.

Need for Action Rules or Preferences In certain cases it is the case that one would like to be able to introduce the concepts of precedence between rules or, alternatively a notion of side effects that are associated to rules.

We define next a conflict that can occur when these are not somehow modelled into the framework.

Definition 9.4.3 (Need for Action Rules or Preferences) *Let $T = \langle D, W \rangle$, where $D = \{\frac{:\neg B}{A}\}$ and $W = \{A \rightarrow B\}$ be a default theory. In T, conflicts occur between the justifications of used defaults and the consequences of formulae in W.* ▲

Example 9.4.3 *To solve the problem in the previous example, the company wants to automatically register every person that it knows as being a candidate if it is not an employee. Operating in a difficult market, it wants further that every candidate (after passing a certain triage) to become an employee. The following is an example of an ASP program reflecting this conflict:*

$$P_3 = \left\{ \begin{array}{l} candidate(X) \leftarrow person(X),\ not\ employee(X). \\ employee(X) \leftarrow person(X),\ candidate(X),\ triage(X). \end{array} \right\}$$

Where the following is its translation to $\mathcal{M}^P$:

$$P_3^{\mathcal{M}^P} = \left\{ \begin{array}{l} pca(ds, X, candidate, generic) \leftarrow not\ pca(ds, X, employee, generic). \\ pca(ds, X, employee, generic) \leftarrow pca(ds, X, candidate, generic). \end{array} \right\}$$

△

Considering the $\mathcal{M}^P$ meta-model, Example 9.4.3 presents a conflict because it has no answer sets. This is typically solved with the introduction of action languages such as the ones described in the introduction that, for the aforementioned rule:

$$candidate(X) \leftarrow person(X),\ not\ employee(X).$$

enforcing $candidate(X)$ as a side effect.

Assigning a higher priority to such rule would also be a potential solution to this problem.

Incoherences: Conflicts can also be defined as incoherencies as we presented in Chapter 7, in the sense of Eiter *et al.* (2010b), in terms of odd negative loops such as the well known barber's paradox.

Definition 9.4.4 (Incoherence: Odd negative loops) *Let $T = \langle D, W \rangle$, where $D = \{\frac{:\neg A}{A}\}$ and $W = \{\}$ be a default theory. In T, incoherence results because there is a dependency between the justifications of used defaults and the consequents of used defaults).* ▲

Example 9.4.4 *The following is an example of an incoherent ASP program reflecting this conflict:*

$$P_4 = \left\{ \begin{array}{l} employee(pedro). \\ allow\_entrance(X) \leftarrow employee(X),\ not\ allow\_entrance(X). \end{array} \right\}$$

Where the following is its $\mathcal{M}^P$ description:

$$P_4^{\mathcal{M}^P} = \left\{ \begin{array}{l} pca(ds, pedro, employee, generic). \\ arca(ds, enter, premises, X, generic) \leftarrow\ pca(ds, X, employee, generic), \\ not\ arca(ds, enter, premises, X, generic). \end{array} \right\}$$

△

We consider the program in Example 9.4.4 to be incoherence and, as such, it has no answer sets.

9.5 Conflict resolution methods

Access control policies are expressed by means of rules which enforce derivation of not only authorisations, access control and integrity constraint checking but also conflict resolution. To resolve rule conflicts, there must be a method for unambiguously choosing a decision. Most conflict resolution methods in practice choose one of the rules in conflict to take precedence over the others. (Other methods are possible, however, such as "majority rules" – choosing the decision of the majority of the rules in conflict).

We list several possible conflict resolution methods below. Note that it is sufficient to define them in terms of their behaviour when exactly two rules are in conflict, because the access control system can handle cases of more than two rules in conflict by following a simple algorithm that does paired matches of each Allow rule against each Deny rule. This algorithm issues an Allow decision if any Allow rule wins its matches against every Deny rule, and otherwise issues a Deny decision. Some of the possible conflict resolution methods for choosing rules to take precedence are:

- Specificity precedence: A rule that applies to a more specific entity takes precedence over a rule that applies to a more general entity.

- Deny precedence: Deny rules take precedence over Allow rules.

- Order precedence: Rules are totally ordered, so it is possible to explicitly state which rules take precedence over others.

- Recency precedence: Rules specified more recently in time take precedence over others. Note that recency precedence is equivalent to order precedence where order is determined by the time at which each rule was set.

These conflict resolution methods may be used in combination. It is possible to use different conflict resolution methods depending on whether conflicting rules differ in the principles they cover, the resources they cover, or both. For example deny precedence if conflicting rules differ in principals, but specificity precedence if conflicting rules differ in resources or in both resources and principals. It may also be necessary to resort to multiple conflict resolution methods when one method fails to resolve a conflict. For example, when conflicting rules cover groups, but those groups are peers of each other, specificity precedence cannot resolve the conflict.

It has been shown in the literature how to use prioritised logic programming to solve authorisation conflicts e.g., Bai (2007), where the authors assign each rule a name representing its preference ordering, using a fixed point semantics to delete those less preferred rules, then using ASP to evaluate the authorisation domain to get the preferred authorisations.

In Ahn *et al.* (2010) different combining algorithms have been identified: "Permit-overrides", "Deny-Overrides", "First-Applicable", and "Only-One-Applicable" as well as the way they can be implemented in ASP.

9.6 Conclusions and Future Work

We identified different types of basic conflicts that occur in access control programs and characterise them in terms of the notion of Relativised Strong Equivalence of logic programs. We also identify conflicts that occur when we introduce default negation and characterise them in terms of default logic while using meta-model $\mathcal{M}^P$ as well as answer set programming throughout that section give examples of those conflicts.

These characterisations enable the detection of conflicts to be done automatically by using automatic theorem provers such as Zinn & Intelligenz (n.d.) and most importantly the ones identified in Cabalar & Lorenzo (2004) where it is stated that the relation they established between S4F and the logic of Here-and-There, allows using modal S4F provers for proving theorems in that intermediate logic. Because of the characterisation of Strongly Equivalent programs as programs that are equivalent in the logic of HT, we can use these theorem provers to perform reasoning and automatically identify the conflicts we characterised before in terms of Strong Equivalence and Relativised Strong Equivalence.

Overall, these characterisations are flexible enough to be extended to several types of conflicts, and can be used to detect which types of conflicts are generated, as well as trace them back to the source (potentially identifying leaks in ACP).

Future Work Introducing strong negation ($\neg$) may lead to modality conflicts, even if this matter has been thoroughly studied and partially solved in the literature through the introduction of paraconsistent semantics and by dealing with this form of negation syntactically.

We still need to investigate the possibility of having a characterisation in the logic of HT or in terms of (relativised) strong equivalence for the conflict types that we identified as occurring with the introduction of default negation.

Research must be done next on conflict resolution methods, formally defining rule combining algorithms in $\mathcal{M}^P$. We also plan to study the implication of using paracoherent semantics such as Semi-Equilibrium models which we presented in Chapter 7. The development of access control policies could also greatly benefit from the work we presented in Chapter 8 on how to calculate justification and debugging models for ASP.

Part IV

Conclusions and Future Work

Chapter 10

Conclusions and Future Work

In this final Chapter 10, we present general conclusions for the work we developed in this thesis as well as directions for future research. This chapter is structured as follows: First, Section 10.1 contains a brief summary of the thesis, including a description of the research problems we dealt with and it is the point where the overall conclusions are provided, part by part. Afterwards, some directions for future work are discussed in Section 10.2, specifically first for the work we presented on modularity of answer set programs and next on ways to deal with conflicts in logic programming.

10.1 Summary and Conclusions

Results Summary In short, the output of this PhD thesis is twofold and consists of

(i) A logic programming framework generalising modular logic programming results and also providing operators for combining arbitrary LP modules, computational complexity results, and a compositional semantics for this generalised modular logic programming framework. We also discussed a probabilistic extension to modular logic programming.

(ii) A characterisation of conflicts that occur when composing modules and ways of dealing with them semantically (by providing a paracoherent semantics) and syntactically by unifying complementary approaches that offer both justifications and debugging models for LP modules. Furthermore, we also provide a prototypical tool for the last part of this work.

With the aforementioned increasing interest around modular logic programming frameworks, a means of dealing with different degrees of certainty as well as to manage conflicts that imply inconsistent and incoherent knowledge is also becoming increasingly important in the field both in academic research and in the industry. As we discussed previously, the way of formally dealing with modularity aspects has already become a very important research topic in the logic programming community. In the

context of modular logic programming (MLP), we lifted the restriction disallowing common outputs in Damásio & Moura (2014) as well as the one forbidding mutual dependencies between modules in Damásio & Moura (2015), for which a thoroughly improved and corrected version is presented in Chapter 5 of this thesis.

Being the case that our first contributions reside on showing how one can combine generalised logic programming modules in the ASP setting, we furthermore characterised conflicts in this setting and are able to find justifications for why a wanted interpretation is not a model and why an unwanted interpretation is a model, thus providing a means to deal with the conflicts syntactically.

This shows the potential for allowing the development of an access control mechanism for dealing with declarative policies — an already very well studied and much needed real-world problem that still is, to a great extent, in need of a solution — that allows to efficiently reason about these policies while providing means that help managing them and the conflicts that arise when they are combined.

10.1.1 Conclusions

In Part II we redefined the necessary modular logic programming operators in order to relax the conditions for combining modules with common atoms in their output signatures as well as to allow arbitrary cyclic dependencies between modules.

For lifting the first restriction, two alternative solutions were presented, both allowing us to retain compositionality while dealing with a more general setting than before.

As for the second restriction, being more complex in nature, we present a model join operation that requires one to look at every model of two modules being composed in order to check for minimality of models that are comparable on account of their inputs. This operation is able to distinguish between atoms that are self supported through positive loops and atoms with proper support. However, this approach is not local as it requires comparing every compatible combination of models and, as it is not general because it does not allow combining modules with integrity constraints, it is of limited applicability. We presented an alternative solution requiring the introduction of extra information in the models for one to be able to detect dependencies. We use models annotated with the way they depend on the atoms in their module's input signature. We then define their semantics in terms of a fixed point operator. The join operator needed then to be redefined and positive dependencies of literals are added to their respective models. This approach turned out to be local, in the sense that we only need to look at the two models being joined and unlike the first alternative we presented, it works well with integrity constraints.

Afterwards, we presented the first approach in the literature to modularise P-log programs and to allow their incremental composition by combining compatible possible worlds and multiplying corresponding unnormalised conditional probability measures clarifying and improving the relationship of P-log with traditional Bayesian Net-

work approaches. We reduce the space and time necessary to make inference in P-log, in contrast with previous algorithms Anh *et al.* (2008); Gelfond *et al.* (2006) which require always enumeration of the full possible worlds (which are exponential on the number of random variables) and repeat calculations.

In Part III For dealing with conflicts that occur in LPs in a semantic way, we provided a semantic characterisation of semi-stable models in terms of bi-models, and of semi-equilibrium models, which eliminate some anomalies of semi-stable models, in terms of HT-models. Furthermore, we characterised the complexity of major reasoning tasks of these semantics.

Regarding implementation, we developed experimental prototypes for computing $\mathcal{SST}(P)$ and $\mathcal{SEQ}(P)$ based on these characterisations. They construct the bi-models (respectively, HT-models) of P and filter them according to the conditions in Theorem 7.3.1 (respectively, Theorem 7.4.1). Alternatively, $\mathcal{SST}(P)$ and $\mathcal{SEQ}(P)$ are obtainable by postprocessing the answer sets of the epistemic transformation P^K respectively its extension P^{HT}, which can be computed with any ASP solver.

As for dealing with conflicts that occur in LPs in a syntactical way, we provided a transformation to compute provenance models under the WF and answer set semantics by computing the answer sets of meta-programs capturing the original programs and then including some necessary extra atoms. We do this in a modular way and preserve compatibility with the previous work of Viegas Damásio *et al.* (2013) and are able to compute these models directly without first obtaining the provenance formulas for certain interpretations. This enables computing provenance answer sets in an easy way by using AS solvers. We then align provenance and debugging answer sets in a unified transformation and show that the provenance approach generalises the debugging one, since any error has a counterpart provenance but not the other way around. Since the proposed method is based on meta-programming, we extended an existing tool Gebser *et al.* (2007b) and developed a proof-of-concept (http://cptkirk.sourceforge.net) tool.

We then presented our more applied work on identifying different types of basic conflicts occurring in access control programs and characterise them in terms of the notion of relativised strong equivalence of logic programs. We also identify conflicts that occur when we introduce default negation and characterise them in terms of default logic while using meta-model $\mathcal{M}^P$ as well as answer set programming throughout that section give examples of those conflicts. Overall, these characterisations are flexible enough to be extended to several types of conflicts, and can be used to detect which types of conflicts are generated, as well as trace them back to the source (potentially identifying leaks in ACP).

10.2 Future Work

Some topics that we leave for future work are natural extensions to the work we developed thus far while some other will have a more exploratory flavour to it. Starting with the more exploratory, studying the implications of allowing negation in the heads of modules and comparing this to evolving logic programs would be very interesting and so would be studying the usage of compositionality in the context of stream reasoning.

10.2.0.1 Future Work on Generalised Modular Logic Programming (Part II)

We laid the fundaments for an integrated modular logic programming framework but still need to relate all the isolated components. We believe it is easy to anticipate that several months or even a few years of work must be put into this tasks and have to leave it for future work. We enunciate a couple of sub-topics of interest: providing modular justifications, and characterising conflicts in GMLP.

As regards future work on the generalised modular logic programming we presented in Chapters 4 and 5, we can straightforwardly extend our results to probabilistic reasoning with answer sets by applying the new module theorem to our work in Damásio & Moura (2011) (presented in Chapter 6), as well as to DLP functions and general stable models. An implementation of the framework is also foreseen in order to assess the overhead when compared with the original benchmarks in Oikarinen & Janhunen (2008). Based on our own preliminary work and results in the literature, we believe that a fully compositional semantics can be attained by resorting to partial interpretations e.g., SE-models Turner (2003b) for defining program models at the semantic level. It is known that one must include extra information about the support of each atom in the models in order to attain generalised compositionality and SE-models appear to be enough.

On the modular probabilistic logic programming track we presented in Chapter 6, as mentioned before, we intend to fully describe the inference algorithm obtained from the compositional semantics of P-log modules and relate it formally with the variable elimination algorithm. We expect that the notion of P-log modules may also help to devise approximate inference methods, e.g., by extending sampling algorithms, enlarging the applicability of P-log which is currently somehow restricted. An interesting road to follow is to generalise the P-log language so as to consider other forms of representing uncertainty like belief functions, possibility measures or even plausibility measures Fagin & Halpern (1994). Applying GMLP to our modular P-Log setting appears to be trivial and we also leave it for future work.

10.2.0.2 Future Work on Conflicts in Modular Logic Programming (Part III)

Concerning future work on the paracoherency track we discussed in Chapter 7, there are several issues. In that Chapter, we considered paracoherence based on program

transformation, as introduced by Sakama & Inoue (1995a). Other notions, like forward chaining construction and strong compatibility Wang *et al.* (2009); Marek *et al.* (1999) might be other candidates to deal with paracoherent reasoning in logic programs, it remains to explore these approaches further.

Another subject is to extend paracoherence to language extensions, including aggregates, nested logic programs etc. Of particular interest are here modular logic programs Janhunen *et al.* (2009); Dao-Tran *et al.* (2009), where module interaction may lead to incoherence. Related to the latter are the more general multi-context systems Brewka & Eiter (2007a), in which knowledge bases exchange beliefs via non-monotonic bridge rules. Based on ideas and results of this thesis, paracoherent semantics for certain classes of such multi-context systems may be devised.

Another issue is to investigate the use of paracoherent semantics in AI applications such as diagnosis, where assumptions may be exploited to generate candidate diagnoses, in the vein of the generalised stable model semantics Kakas & Mancarella (1990).

Also, in Pereira *et al.* (1993b), the authors apply a contradiction removal approach to normal logic programs and use it uniformly to treat diagnosis and debugging.

Finally, a promising line of work is to apply the transformation used on the semi-equilibrium semantics and check if it works well with modular compositionality, having in consideration disjunctive MLPs in as much as the $\mathcal{SEQ}$ transformation is also disjunctive.

The mapping between debugging and justification models we presented in Chapter 8 allows generating answer sets capturing errors and justifications for (intended) models. As expected, they are exponential. One direction to explore is to obtain prime implicant by optimising these models using reification and then subset inclusion preference ordering Gebser *et al.* (2007a, 2011b) via a saturation technique Eiter & Gottlob (1995b). Note that deciding if an AS is optimal for a DLP is a Π_2^p-complete problem.

As for the work we presented in Chapter 9 on conflicts in ACP, as we mentioned before, the introduction of strong negation ($\neg$) may lead to modality conflicts, even if this matter has been thoroughly studied and partially solved in the literature through the introduction of paraconsistent semantics and by dealing with it syntactically.

We still need to investigate the possibility of having a characterisation in the logic of HT or in terms of (relativised) strong equivalence for the conflict types that we identified as occurring with the introduction of default negation.

Research must be done next on conflict resolution methods, formally defining rule combining algorithms in $\mathcal{M}^P$. We also plan to study the implication of using paracoherent semantics such as semi-equilibrium models which was presented in Eiter *et al.* (2010b). It is necessary also to investigate the usage of action languages to solve problems that arise from the introduction of default negation.

Finally, another issue is to investigate the requirements to generalise these charac-

terisations of conflicts to our generalised modular logic programming (GMLP) setting.

Index

Bibliography

Ahn, Gail-Joon, Hu, Hongxin, Lee, Joohyung, & Meng, Yunsong. 2010. Representing and Reasoning about Web Access Control Policies. *Pages 137–146 of:* Ahamed, Sheikh Iqbal, Bae, Doo-Hwan, Cha, Sung Deok, Chang, Carl K., Subramanyan, Rajesh, Wong, Eric, & Yang, Hen-I (eds), *COMPSAC*. IEEE Computer Society. page 180

Al-Kahtani, Mohammad A., & Sandhu, Ravi. 2004. Rule-Based RBAC with Negative Authorization. *Pages 405–415 of: Proceedings of the 20th Annual Computer Security Applications Conference*. ACSAC '04. Washington, DC, USA: IEEE Computer Society. page 166

Alcântara, Joao, Damásio, Carlos Viegas, & Pereira, Luís Moniz. 2005. A Declarative Characterisation of Disjunctive Paraconsistent Answer Sets. *Pages 915–952 of: Proceedings of 16th European Conference on Artificial Intelligence (2004), Valencia, Spain (22nd–27th September 2004)*. page 18, page 114, page 136

Amendola, Giovanni, Eiter, Thomas, & Leone, Nicola. 2014. Modular Paracoherent Answer Sets. *Pages 457–471 of:* Fermé, Eduardo, & Leite, João (eds), *Logics in Artificial Intelligence - 14th European Conference, JELIA 2014, Funchal, Madeira, Portugal, September 24-26, 2014. Proceedings*. Lecture Notes in Computer Science, vol. 8761. Springer. page 138

Amendola, Giovanni, Eiter, Thomas, Fink, Michael, Leone, Nicola, & Moura, Joao. 2015. Semi-Equilibrium Models for Paracoherent Answer Set Programs. *In: Technical report, final version to appear*. page 9, page 31, page 138

Anh, Han The, Kencana Ramli, Carroline D., & Damásio, Carlos Viegas. 2008. An Implementation of Extended P-Log Using XASP. *Pages 739–743 of: ICLP '08: Proceedings of the 24th International Conference on Logic Programming*. Berlin, Heidelberg: Springer-Verlag. page 87, page 102, page 189

Artemov, Sergei N. 1995. Operational modal logic. page 142

Babb, Joseph, & Lee, Joohyung. 2012. Module theorem for the general theory of stable models. *TPLP*, **12**(4-5), 719–735. page 21, page 44, page 57

Bai, Yun. 2007. Logic Program for Authorizations. *World Academy of Science, Engineering and Technology issue 33*. page 180

Balduccini, Marcello, & Gelfond, Michael. 2003. Logic Programs with Consistency-Restoring Rules. *Pages 9–18 of: International Symposium on Logical Formalization of Commonsense Reasoning, AAAI 2003 Spring Symposium Series*. page 32

Baral, Chitta. 2003. *Knowledge Representation, Reasoning, and Declarative Problem Solving*. Cambridge University Press. page 8, page 43

Baral, Chitta, & Hunsaker, Matt. 2007. Using the probabilistic logic programming language P-log for causal and counterfactual reasoning and non-naive conditioning. *Pages 243–249 of: Proceedings of the 20th international joint conference on Artifical intelligence*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. page 30, page 87

Baral, Chitta, Gelfond, Michael, & Rushton, Nelson. 2004. Probabilistic reasoning with answer sets. *Pages 21–33 of: In Proceedings of LPNMR-7*. Springer. page 30, page 87, page 88, page 92, page 94

Barker, Steve. 2009. The next 700 access control models or a unifying meta-model? *Pages 187–196 of: Proceedings of the 14th ACM symposium on Access control models and technologies*. SACMAT '09. New York, NY, USA: ACM. page 35, page 169, page 171

Barker, Steve. 2010. Personalizing access control by generalizing access control. *Pages 149–158 of: Proceedings of the 15th ACM symposium on Access control models and technologies*. SACMAT '10. New York, NY, USA: ACM. page 165, page 169, page 171

Bertino, Elisa. 2005. PDL with preferences. *Pages 213–222 of: Proc. of POLICY*. page 175

Bonatti, Piero, De Capitani di Vimercati, Sabrina, & Samarati, Pierangela. 2002. An Algebra for Composing Access Control Policies. *ACM Trans. Inf. Syst. Secur.*, **5**(1), 1–35. page 8, page 34

Bonatti, Piero A., Coi, Juri Luca De, Olmedilla, Daniel, & Sauro, Luigi. 2009. Rule-Based Policy Representations and Reasoning. *Pages 201–232 of:* Bry, François, & Maluszynski, Jan (eds), *REWERSE*. Lecture Notes in Computer Science, vol. 5500. Springer. page 34, page 167

Brain, Martin, Gebser, Martin, Pührer, Jörg, Schaub, Torsten, Tompits, Hans, & Woltran, Stefan. 2007. Debugging ASP Programs by Means of ASP. *Pages 31–43 of:* Baral, Chitta, Brewka, Gerhard, & Schlipf, John S. (eds), *LPNMR 2007*. Lecture Notes in Computer Science, vol. 4483. Springer. page 142, page 143

Brewka, Gerd, & Eiter, Thomas. 2007a. Equilibria in Heterogeneous Nonmonotonic Multi-Context Systems. *Pages 385–390 of: Proceedings 22nd Conference on Artificial Intelligence (AAAI '07), July 22-26, 2007, Vancouver*. AAAI Press. page 139, page 191

Brewka, Gerhard, & Eiter, Thomas. 2007b. Equilibria in heterogeneous nonmonotonic multi-context systems. *Pages 385–390 of: AAAI*, vol. 7. page 20, page 79, page 80, page 81

Brewka, Gerhard, Eiter, Thomas, & Truszczyński, Miroslaw. 2011. Answer Set Programming at a Glance. *Commun. ACM*, **54**(12), 92–103. page 8

Brezhnev, Vladimir [N.]. 2001 (Aug.). On the Logic of Proofs. *Pages 35–46 of:* Striegnitz, Kristina (ed), *Proceedings of the sixth ESSLLI Student Session, 13th European Summer School in Logic, Language and Information (ESSLLI'01)*. FoLLI, Helsinki. page 142

Bugliesi, Michele, Lamma, Evelina, & Mello, Paola. 1994. Modularity in Logic Programming. *J. Log. Program.*, **19/20**, 443–502. page 44

Busoniu, Paula-Andra, Oetsch, Johannes, PUHRER, JORG, SKOCOVSKI, PETER, & TOMPITS, HANS. 2013. SeaLion: An eclipse-based IDE for answer-set programming with advanced debugging support. *Theory and Practice of Logic Programming*, **13**(4-5), 657–673. page 142

Cabalar, Pedro. 2011. Logic Programs and Causal Proofs. *In: Logical Formalizations of Commonsense Reasoning, Papers from the 2011 AAAI Spring Symposium, Technical Report SS-11-06, Stanford, California, USA, March 21-23, 2011*. AAAI. page 142, page 151

Cabalar, Pedro, & Fandiño, Jorge. 2013. An Algebra of Causal Chains. *Pages 530–542 of:* Cabalar, Pedro, & Son, TranCao (eds), *Logic Programming and Nonmonotonic Reasoning*. Lecture Notes in Computer Science, vol. 8148. Springer Berlin Heidelberg. page 151

Cabalar, Pedro, & Fandinno, Jorge. 2017. Enablers and inhibitors in causal justifications of logic programs. *Theory and Practice of Logic Programming*, **17**(1), 49–74. page 142, page 151, page 159, page 160

Cabalar, Pedro, & Lorenzo, David. 2004. New Insights on the Intuitionistic Interpretation of Default Logic. *Pages 798–802 of:* de Mántaras, Ramon López, & Saitta, Lorenza (eds), *ECAI*. IOS Press. page 176, page 180

Cabalar, Pedro, Odintsov, Sergei P., Pearce, David, & Valverde, Agustín. 2007. Partial equilibrium logic. *Ann. Math. Artif. Intell.*, **50**(3-4), 305–331. page 135, page 136

Cabalar, Pedro, Fandinno, Jorge, & Fink, Michael. 2014. Causal Graph Justifications of Logic Programs. *TPLP*, **14**(4-5), 603–618. page 142, page 151

Damásio, Carlos Viegas, & Moura, João. 2011. Modularity of P-Log Programs. *Pages 13–25 of:* Delgrande, James P., & Faber, Wolfgang (eds), *LPNMR*. Lecture Notes in Computer Science, vol. 6645. Springer. page 10, page 21, page 30, page 58, page 190

Damásio, Carlos Viegas, & Moura, João. 2014. Generalizing Modular Logic Programs. *CoRR - Proceedings of NMR 2014-15th International Workshop on Non-Monotonic Reasoning*, **abs/1404.7205**. page 10, page 21, page 30, page 188

Damásio, Carlos Viegas, & Moura, João. 2015 (September). Allowing Cyclic Dependencies in Modular Logic Programming. *In: EPIA 2015 – 17th Portuguese Conference on Artificial Intelligence*. page 10, page 30, page 188

Damásio, Carlos Viegas, Moura, João, & Analyti, Anastasia. 2015. Unifying Justifications and Debugging for Answer-Set Programs. *31st International Conference on Logic Programming (ICLP 2015)*. page 10, page 33

Dao-Tran, Minh, Eiter, Thomas, Fink, Michael, & Krennwallner, Thomas. 2009. Modular Nonmonotonic Logic Programming Revisited. *Pages 145–159 of:* Hill, Patricia M., & Warren, David S. (eds), *25th International Conference on Logic Programming (ICLP 2009), Pasadena, California, USA, July 14–17, 2009*. LNCS, vol. 5649. Springer. page 21, page 44, page 45, page 54, page 57, page 59, page 85

Dao-Tran, Minh, Eiter, Thomas, Fink, Michael, & Krennwallner, Thomas. 2009. Modular Nonmonotonic Logic Programming Revisited. *Pages 145–159 of:* Hill, P.M., & Warren, D.S. (eds), *Proceedings 25th International Conference on Logic Programming (ICLP 2009)*. Lecture Notes in Computer Science, no. 5649. Springer. page 139, page 191

Eiter, T., & Gottlob, G. 1995a. On the Computational Cost of Disjunctive Logic Programming: Propositional Case. *Annals of Mathematics and Artificial Intelligence*, **15**(3/4), 289–323. page 134

Eiter, T., Ianni, G., Polleres, A., & Schidlauer, R. June 2006. *Answer Set Programming for the Semantic Web*. page 168

Eiter, Thomas, & Gottlob, Georg. 1995b. *On the Computational Cost of Disjunctive Logic Programming: Propositional Case*. page 164, page 191

Eiter, Thomas, Leone, Nicola, & Sacca, Domenico. 1997a. *On the Partial Semantics for Disjunctive Deductive Databases*. page 19

Eiter, Thomas, Leone, Nicola, & Saccà, Domenico. 1997b. On the Partial Semantics for Disjunctive Deductive Databases. *Annals of Mathematics and Artificial Intelligence*, **19**(1/2), 59–96. page 114, page 134, page 135, page 136

Eiter, Thomas, Faber, Wolfgang, Leone, Nicola, & Pfeifer, Gerald. 2001. Computing Preferred and Weakly Preferred Answer Sets by Meta-Interpretation in Answer Set Programming. *Pages 45–52 of: Proceedings AAAI 2001 Spring Symposium on Answer Set Programming*. AAAI Press. page 8, page 43

Eiter, Thomas, Fink, Michael, Tompits, Hans, & Woltran, Stefan. 2004. Simplifying logic programs under uniform and strong equivalence. *Pages 87–99 of: In LPNMR'2004*. Springer. page 175

Eiter, Thomas, Fink, Michael, & Woltran, Stefan. 2007. Semantical characterizations and complexity of equivalences in answer set programming. *ACM Trans. Comput. Logic*, **8**(3). page 111, page 166, page 172, page 173

Eiter, Thomas, Fink, Michael, Schüller, Peter, & Weinzierl, Antonius. 2010a. Finding Explanations of Inconsistency in Multi-Context Systems. *KR*, **10**, 329–339. page 7, page 33, page 143

Eiter, Thomas, Fink, Michael, & Moura, Joao. 2010b. Paracoherent Answer Set Programming. *In:* Lin, Fangzhen, Sattler, Ulrike, & Truszczynski, Miroslaw (eds), *Principles of Knowledge Representation and Reasoning: Proceedings of the Twelfth International Conference, KR 2010, Toronto, Ontario, Canada, May 9-13, 2010*. AAAI Press. page 9, page 31, page 132, page 179, page 191

Fagin, Ronald, & Halpern, Joseph Y. 1994. Reasoning About Knowledge and Probability. *J. ACM*, **41**(2), 340–367. page 102, page 190

Ferraris, Paolo, & Lifschitz, Vladimir. 2005. Weight constraints as nested expressions. *TPLP*, **5**(1-2), 45–74. page 12

Ferraris, Paolo, Lee, Joohyung, & Lifschitz, Vladimir. 2007. A new perspective on stable models. *Pages 372–379 of: Proceedings of the 20th international joint conference on Artifical intelligence*. IJCAI'07. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. page 33, page 143

Fitting, Melvin. 2005. The logic of proofs, semantically. *Annals of Pure and Applied Logic*, **132**(1), 1–25. page 142

Gaifman, H., & Shapiro, E. 1989. Fully abstract compositional semantics for logic programs. *Pages 134–142 of: POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM. page 21, page 41, page 43, page 44, page 45

Gebser, Martin, Kaufmann, Benjamin, Neumann, André, & Schaub, Torsten. 2007a. clasp: A conflict-driven answer set solver. *Pages 260–265 of: Logic Programming and Nonmonotonic Reasoning*. Springer Berlin Heidelberg. page 164, page 191

Gebser, Martin, Pührer, Jörg, Schaub, Torsten, Tompits, Hans, & Woltran, Stefan. 2007b. spock: A Debugging Support Tool for Logic Programs under the Answer-Set Semantics. *Pages 258–261 of:* Seipel, Dietmar, Hanus, Michael, Wolf, Armin, & Baumeister, Joachim (eds), *Proceedings of the 21st Workshop on (Constraint) Logic Programming, (WLP'07), Würzburg, Germany*. Technical Report 434, Bayerische Julius-Maximilians-Universität Würzburg, Institut für Informatik. page 33, page 164, page 189

Gebser, Martin, Puehrer, Joerg, Schaub, Torsten, & Tompits, Hans. 2008. A Meta-Programming Technique for Debugging Answer-Set Programs. *Pages 448–453 of:* Fox, Dieter, & Gomes, Carla P. (eds), *AAAI-08/IAAI-08 Proceedings*. page 7, page 32, page 33, page 141, page 143, page 145, page 146, page 150, page 152, page 161, page 162

Gebser, Martin, Grote, Torsten, Kaminski, Roland, & Schaub, Torsten. 2011a. Reactive answer set programming. *Pages 54–66 of: Proceedings of the 11th international conference on Logic programming and nonmonotonic reasoning*. LPNMR'11. Berlin, Heidelberg: Springer-Verlag. page 58

Gebser, Marting, Kaminski, Roland, & Schaub, Torsten. 2011b. Complex optimization in answer set programming. *Theory and Practice of Logic Programming*, **11**(7), 821–839. page 164, page 191

Gelfond, M., Rushton, N., & Zhu, W. 2006. Combining logical and probabilistic reasoning. *Pages 50–55 of: Proc. of AAAI 06 Spring Symposium: Formalizing and Compiling Background Knowledge and Its Applications to Knowledge Representation and Question Answering*. AAAI Press. page 87, page 102, page 189

Gelfond, Michael, & Lifschitz, Vladimir. 1988. The Stable Model Semantics For Logic Programming. MIT Press. page 8, page 14, page 20, page 30, page 79, page 87, page 165

Gelfond, Michael, & Lifschitz, Vladimir. 1990. Logic Programs with Classical Negation. *Pages 579–597 of:* Warren, David H. D., & Szeredi, Péter (eds), *Logic Programming, Proceedings of the Seventh International Conference, Jerusalem, Israel, June 18-20, 1990*. MIT Press. page 14, page 87

Giordano, Laura, & Martelli, Alberto. 1994. Structuring logic programs: a modal approach. *The Journal of Logic Programming*, **21**(2), 59 – 94. page 44

Heyting, A. 1930. Die formalen Regeln der intuitionistischen Logik. *Sitz*, **Berlin**, 42–56. page 15

Janhunen, Tomi, Oikarinen, Emilia, Tompits, Hans, & Woltran, Stefan. 2009. Modularity aspects of disjunctive stable models. *J. Artif. Int. Res.*, **35**(1), 813–857. page v, page vii, page 6, page 21, page 57, page 139, page 191

Järvisalo, Matti, Oikarinen, Emilia, Janhunen, Tomi, & Niemelä, Ilkka. 2009. A Module-Based Framework for Multi-Language Constraint Modeling. *Pages 155–169 of:* Erdem, Esra, Lin, Fangzhen, & Schaub, Torsten (eds), *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2009)*. Lecture Notes in Artificial Intelligence, vol. 5753. Springer. page 21, page 44, page 45

Kakas, Antonis C., & Mancarella, Paolo. 1990. Generalized Stable Models: A Semantics for Abduction. *Pages 385–391 of: ECAI*. page 139, page 191

Kolovski, Vladimir. 2007. *Logic-Based Access Control Policy Specification and Management*. page 34, page 167

Kwiatkowska, M., Norman, G., & Parker, D. 2001 (September). PRISM: Probabilistic Symbolic Model Checker. *Pages 7–12 of:* Kemper, P. (ed), *Proc. Tools Session of Aachen 2001 International Multiconference on Measurement, Modelling and Evaluation of Computer-Communication Systems*. page 87

Lee, Joohyung. 2005. A Model-Theoretic Counterpart of Loop Formulas. *Pages 503–508 of: Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*. Professional Book Center. page 33, page 143

Lifschitz, Vladimir. 1999. Action Languages, Answer Sets and Planning. *Pages 357–373 of: In The Logic Programming Paradigm: a 25-Year Perspective*. Springer Verlag. page 14

Lifschitz, Vladimir. 2002. Answer set programming and plan generation. *Artificial Intelligence*, **138**(1-2), 39–54. page 8, page 43

Lifschitz, Vladimir. 2008a. Twelve Definitions of a Stable Model. *Pages 37–51 of: Proceedings of International Conference on Logic Programming (ICLP)*. page 87

Lifschitz, Vladimir. 2008b. What Is Answer Set Programming? *Pages 1594–1597 of: Proceedings of the AAAI Conference on Artificial Intelligence*. MIT Press. page 87

Lifschitz, Vladimir, & Turner, Hudson. 1994. Splitting a Logic Program. *Pages 23–37 of: Proceedings of the Eleventh International Conference on Logic Programming*. Cambridge, MA, USA: MIT Press. page 43, page 57

Lifschitz, Vladimir, Tang, Lappoon R., & Turner, Hudson. 1999. Nested Expressions in Logic Programs. *Annals of Mathematics and Artificial Intelligence*, **25**, 369–389. page 172

Lifschitz, Vladimir, Pearce, David, & Valverde, Agustin. 2000. Strongly Equivalent Logic Programs. *ACM Transactions on Computational Logic*, **2**, 2001. page 74, page 111, page 166, page 172

Mancarella, Paolo, & Pedreschi, Dino. 1988. An Algebra of Logic Programs. *Pages 1006–1023 of: ICLP/SLP*. page 44

Marek, V. Wiktor, Nerode, Anil, & Remmel, Jeffrey B. 1999. Logic Programs, Well-Orderings, and Forward Chaining. *Annals of Pure and Applied Logic*, **96**(1-3), 231–276. page 138, page 191

Marek, Victor W., & Truszczynski, Miroslaw. 1999. Stable Models and an Alternative Logic Programming Paradigm. *In: The Logic Programming Paradigm: a 25-Year Perspective*. page 8, page 43

Mendelson, Elliott. 1987. *Introduction to Mathematical Logic; (3rd Ed.)*. Monterey, CA, USA: Wadsworth and Brooks/Cole Advanced Books & Software. page 11

Miller, Dale. 1986. A Theory of Modules for Logic Programming. *Pages 106–114 of: In Symp. Logic Programming*. page 44

Moura, João. 2012. Characterising Access Control Conflicts. *In: NMR 2012-14th International Workshop on Non-Monotonic Reasoning*. page 10, page 35

Niemelä, Ilkka. 1998. Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm. *Annals of Mathematics and Artificial Intelligence*, **25**, 72–79. page 8, page 12, page 43

Odintsov, Sergei P., & Pearce, David. 2005. Routley Semantics for Answer Sets. *Pages 343–355 of:* Baral, Chitta, Greco, Gianluigi, Leone, Nicola, & Terracina, Giorgio (eds), *LPNMR*. Lecture Notes in Computer Science, vol. 3662. Springer. page 18, page 114, page 136

Oetsch, Johannes, PÜhrer, JÖrg, & Tompits, Hans. 2010. Catching the Ouroboros: On Debugging Non-ground Answer-set Programs. *Theory Pract. Log. Program.*, **10**(4-6), 513–529. page 143

Oikarinen, Emilia, & Janhunen, Tomi. 2008. Achieving compositionality of the stable model semantics for smodels programs1. *Theory Pract. Log. Program.*, **8**(5-6), 717–761. page v, page vii, page 6, page 12, page 21, page 24, page 25, page 29, page 31, page 36, page 41, page 43, page 44, page 45, page 58, page 85, page 88, page 103, page 190

O'Keefe, Richard A. 1985. Towards an Algebra for Constructing Logic Programs. *Pages 152–160 of: SLP*. page 44

Osorio, Mauricio, Ramírez, José R. Arrazola, & Carballido, José Luis. 2008. Logical Weak Completions of Paraconsistent Logics. *J. Log. Comput.*, **18**(6), 913–940. page 19, page 114, page 136

Pearce, David. 2006a. Equilibrium logic. *Annals of Mathematics and Artificial Intelligence*, **47**(1), 3. page 15, page 17

Pearce, David. 2006b. Equilibrium logic. *Ann. Math. Artif. Intell.*, **47**(1-2), 3–41. page 32

Pearce, David, & Valverde, Agustín. 2008. Quantified Equilibrium Logic and Foundations for Answer Set Programs. *Pages 546–560 of:* de la Banda, Maria Garcia, & Pontelli, Enrico (eds), *ICLP*. Lecture Notes in Computer Science, vol. 5366. Springer. page 15, page 17, page 19, page 32, page 114, page 137

Pearl, J. 1988. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann. page 30, page 88

Pearl, J. 2000. *Causality: Models, Reasoning and Inference*. Cambridge University Press. page 30, page 87, page 88

Pemmasani, Giridhar, Guo, Hai-Feng, Dong, Yifei, Ramakrishnan, C.R., & Ramakrishnan, I.V. 2004. Online Justification for Tabled Logic Programs. *Pages 24–38 of:* Kameyama, Yukiyoshi, & Stuckey, PeterJ. (eds), *Functional and Logic Programming*. Lecture Notes in Computer Science, vol. 2998. Springer Berlin Heidelberg. page 7, page 147, page 151

Pereira, L. M., Damásio, C. V., & Alferes, J. J. 1993a. Debugging by Diagnosing Assumptions. *Pages 58–74 of:* Fritzson, P. A. (ed), *Automated and Algorithmic Debugging, First International Workshop (AADEBUG'93)*. Lecture Notes in Computer Science (LNAI), vol. 749. Linköping, Suécia: Springer. page 32

Pereira, Luís Moniz, & Pinto, Alexandre Miguel. 2005. Revised Stable Models - A Semantics for Logic Programs. *Pages 29–42 of:* Bento, Carlos, Cardoso, Amílcar, & Dias, Gaël (eds), *EPIA*. Lecture Notes in Computer Science, vol. 3808. Springer. page 19, page 114, page 136

Pereira, Luís Moniz, Alferes, José Júlio, & Aparício, Joaquim Nunes. 1992. Contradiction Removal Semantics with Explicit Negation. *Pages 91–105 of:* Masuch, Michael, & Pólos, László (eds), *Knowledge Representation and Reasoning Under Uncertainty, Logic at Work [International Conference Logic at Work, Amsterdam, The Netherlands, December 17-19, 1992]*. Lecture Notes in Computer Science, vol. 808. Springer. page 116

Pereira, Luís Moniz, Damásio, Carlos Viegas, & Alferes, José Júlio. 1993b. Diagnosis and Debugging as Contradiction Removal. *Pages 316–330 of: PROCEEDINGS OF THE 2ND INTERNATIONAL WORKSHOP ON LOGIC PROGRAMMING AND NON-MONOTONIC REASONING*. MIT Press. page 143, page 191

Pfeffer, Avi, & Koller, Daphne. 2000. Semantics and Inference for Recursive Probability Models. *Pages 538–544 of: Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*. AAAI Press. page 87

Polleres, Axel, Fruehstueck, Melanie, Schenner, Gottfried, & Friedrich, Gerhard. 2013. Debugging Non-ground ASP Programs with Choice Rules, Cardinality and Weight Constraints. *Pages 452–464 of:* Cabalar, Pedro, & Son, TranCao (eds), *Logic Programming and Nonmonotonic Reasoning*. Lecture Notes in Computer Science, vol. 8148. Springer Berlin Heidelberg. page 143

Pontelli, Enrico, & Son, Tran Cao. 2006. T.: Justifications for logic programs under answer set semantics. *In: Proceedings of the 22nd International Conference on Logic Programming (ICLP ,Äô06). Springer-Verlag (2006) 196 ,Äì210*. Springer. page 151

Pontelli, Enrico, Son, Tran Cao, & El-Khatib, Omar. 2009. Justifications for logic programs under answer set semantics. *TPLP*, **9**(1), 1–56. page 7, page 32, page 147, page 151, page 164

Poole, David. 1997. The independent choice logic for modeling multiple agents under uncertainty. *Artificial Intelligence*, **94**(July), 7–56. page 87

Poole, David L., & Zhang, Nevin Lianwen. 2011. Exploiting Contextual Independence In Probabilistic Inference. *CoRR*, **abs/1106.4864**. page 102

Przymusinski, T. 1991a. Stable Semantics for Disjunctive Programs. *New Generation Computing*, **9**, 401–424. page 114, page 135

Przymusinski, Teodor. 1990. Well-Founded Semantics Coincides with Three-Valued Stable Semantics. *Fundamenta Informaticae*, **13**, 445–463. page 14

Przymusinski, Teodor. 1991b. Three-Valued Non-Monotonic Formalisms And Semantics of Logic Programs. *Artificial Intelligence*, **49**, 341–348. page 14, page 19

Reiter, R. 1987. *A logic for default reasoning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. Pages 68–93. page 176

Routley, Richard. 1974. Semantical Analyses of Propositional Systems of Fitch and Nelson. *Pages 283–298 of: Studia Logica 33*. page 136

Russell, Stuart, & Norvig, Peter. 2010. *Artificial Intelligence: A Modern Approach*. 3 edn. Prentice Hall. page 88

Sakama, Chiaki, & Inoue, Katsumi. 1995a. Paraconsistent Stable Semantics for Extended Disjunctive Programs. *J. Log. Comput.*, **5**(3), 265–285. page 18, page 19, page 110, page 114, page 116, page 117, page 118, page 119, page 136, page 137, page 138, page 191

Sakama, Chiaki, & Inoue, Katsumi. 1995b. Paraconsistent stable semantics for extended disjunctive programs. *Journal of Logic and Computation*, **5**, 265–285. page 19, page 20, page 119

Sato, Taisuke. 1995. A Statistical Learning Method for Logic Programs with Distribution Semantics. *Pages 715–729 of: In proceedings of the 12th international conference on Logic Programming (ICLP'95)*. MIT Press. page 94

Sato, Taisuke, & Kameya, Yoshitaka. 2001. Parameter learning of logic programs for symbolic-statistical modeling. *Journal of Artificial Intelligence Research*, 454. page 94

Shchekotykhin, Kostyantyn M. 2014. Interactive Debugging of ASP Programs. *CoRR*, **abs/1403.5142**. page 7, page 33

Slota, Martin, & Leite, João. 2012. Robust Equivalence Models for Semantic Updates of Answer-Set Programs. *In:* Brewka, Gerhard, Eiter, Thomas, & McIlraith, Sheila A. (eds), *Proc. of KR 2012*. AAAI Press. page 68

Tasharrofi, Shahab, & Ternovska, Eugenia. 2011. A semantic account for modularity in multi-language modelling of search problems. *Pages 259–274 of: Proceedings of the 8th international conference on Frontiers of combining systems*. FroCoS'11. Berlin, Heidelberg: Springer-Verlag. page 44, page 57

Tasharrofi, Shahab, & Ternovska, Eugenia. 2014. Generalized Multi-Context Systems. *In: Principles of Knowledge Representation and Reasoning: Proceedings of the Forteenth International Conference, KR 2014, Vienna, Austria, July, 2014*. KR'14. AAAI Press. page 20, page 79, page 80, page 81, page 82

Turner, Hudson. 2003a. Strong equivalence made easy: nested expressions and weight constraints. *TPLP*, **3**(4-5), 609–622. page 17

Turner, Hudson. 2003b. Strong equivalence made easy: nested expressions and weight constraints. *Theory and Practice of Logic Programming*, **3**(4), 609–622. page 58, page 190

van Gelder, A., Ross, K.A., & Schlipf, J.S. 1991. The Well-Founded Semantics for General Logic Programs. *Journal of the ACM*, **38**(3), 620–650. page 14, page 117, page 136

Viegas Damásio, Carlos, Analyti, Anastasia, & Antoniou, Grigoris. 2013. Justifications for Logic Programming. *Pages 530–542 of:* Cabalar, Pedro, & Son, Tran Cao (eds), *Logic Programming and Nonmonotonic Reasoning*. Lecture Notes in Computer Science, vol. 8148. Springer Berlin Heidelberg. page 28, page 32, page 33, page 110, page 141, page 142, page 143, page 146, page 147, page 149, page 150, page 151, page 159, page 163, page 189

Vos, Marina De, & Vermeir, Dirk. 2000. Dynamically Ordered Probabilistic Choice Logic Programming. *Pages 227–239 of:* Kapoor, Sanjiv, & Prasad, Sanjiva (eds), *Foundations of Software Technology and Theoretical Computer Science, 20th Conference, FST TCS 2000 New Delhi, India, December 13-15, 2000, Proceedings.* Lecture Notes in Computer Science, vol. 1974. Springer. page 95

Wang, Yigong, Zhang, Hongqi, Dai, Xiangdong, & Liu, Jiang. 2010 (dec.). Conflicts analysis and resolution for access control policies. *Pages 264 –267 of: Information Theory and Information Security (ICITIS), 2010 IEEE International Conference on*. page 173

Wang, Yisong, Zhang, Mingyi, & You, Jia-Huai. 2009. Logic Programs, Compatibility and Forward Chaining Construction. *J. Comput. Sci. Technol.*, **24**(6), 1125–1137. page 138, page 191

Woo, Thomas Y. C., & Lam, Simon S. 1992. Authorization in Distributed Systems:A Formal Approach. *Security and Privacy, IEEE Symposium on*, **0**, 33. page 176

You, J.-H., & Yuan, L.Y. 1994. A Three-Valued Semantics for Deductive Databases and Logic Programs. *Journal of Computer and System Sciences*, **49**, 334–361. page 19, page 114, page 134, page 136

Zhang, Nevin Lianwen, & Poole, David. 1996. Exploiting Causal Independence in Bayesian Network Inference. *J. Artif. Int. Res.*, **5**(1), 301–328. page 88, page 99, page 101, page 102

Zinn, Claus, & Intelligenz, Lehrstuhl Fur Kunstliche. *COLOSSEUM - An Automated Theorem Prover for Intuitionistic Predicate Logic based on Dialogue Games*. page 180

Index